City, University of London M.Sc. in Data Science

School of Mathematics, Computing Science & Engineering

Project Report

2019-2021

Automatic Harmonic Analysis: Roman Numerals via Deep Learning

Student: **Brenner Swenson**

Supervisor: **Dr. Tillman Weyde**

Date of Submission: 14/04/2022

Declaration

By submitting this work, I declare that this work is entirely my own except those parts duly identified and referenced in my submission. It complies with any specified word limits and the requirements and regulations detailed in the assessment instructions and any other relevant programme and module documentation. In submitting this work, I acknowledge that I have read and understood the regulations and code regarding academic misconduct, including that relating to plagiarism, as specified in the Programme Handbook. I also acknowledge that this work will be subject to a variety of checks for academic misconduct.

Signed: **Brenner Swenson**

Abstract

This research aimed to develop and investigate new techniques for Automatic Harmonic Analysis (AHA), specifically Roman Numeral Classification (RNC), by combining deep learning techniques from research recently published in the MIR community. This work implements architectural components from performant model architectures like the Transformer (Vaswani et al., 2017). This work sought to investigate how these types of architectures perform outside of natural language processing, develop performant models for RNC, and better understand the relationships between the individual tasks required for AHA.

This project's model utilises the Transformer's encoder architecture with multi-headed self-attention. The model optionally uses relative positional encoding—a technique to increase the importance of nearby tokens when calculating attention scores. This research yielded that an encoder-based multi-task model with self-attention is capable of achieving comparable results to the best existing RNC models. It was determined that the inclusion of relative positional encoding to clip the encoding values beyond a certain distance in both forward and backward directions did not add to overall performance but warrants further investigation.

The Transformer model benefits from larger training dataset sizes, (Popel & Bojar, 2018) and it is believed that the performance of this project's models could benefit greatly from a bigger, more balanced dataset of music annotated with Roman numerals. Future efforts to mitigate class imbalance and develop a more robust data augmentation process are recommended. This work found that overall RNC performance benefits by concatenating the predictions of certain task to the inputs of others, specifically the tasks that indicate global contextual information in a piece of music. This design decision resulted in each of the five Roman numeral tasks' predictions agreeing with one another more frequently, but ultimately did not result in an overall accuracy higher than the benchmark model.

Contents

1. Introduction and Objectives

1.1 Preface

Within the context of Western music theory, **chords**, or multiple notes sounding at the same time, can be labelled in multiple ways; not only by individually listing the notes within them, but by their functional relationship to a larger musical piece. This project seeks to develop a machine learning model that utilises architectural elements from the Transformer model (Vaswani et al., 2017) to automatically annotate chords in classical music with their respective harmonic functions. These functions are communicated via the Roman numeral taxonomy, a system unique to tonal music theory.

It should be noted that this report references and relies upon concepts from music theory. These concepts are briefly explained throughout, often with references to theory textbooks. This was done with the aim of providing context relevant for the project's objectives. The brief harmonic summaries in this paper are not intended to be exhaustive or comprehensive. Rather, they are intended to provide enough information on the project's domain without the majority of this report consisting of pedagogical music theory jargon. The first three chapters of Stephen G. Laitz's music theory textbook *The Complete Musician* are an excellent companion while reading this report. A glossary for the **bolded** music theory terms can be found immediately following the paper's conclusion.

1.2 Roman Numerals and Music

In the theory of tonal music, Roman numerals (RN) provide a framework with which musicians and analysts alike can communicate a musical chord's function and its qualities independent of the musical piece's key. The **key** of a composition is the main set of **pitches**, or notes, that constitutes the piece of music's harmonic foundation. A key is defined by a **root** note, and a **mode**, e.g., major or minor. Keys are why many chord progressions end with a feeling of "returning home" upon resolution.

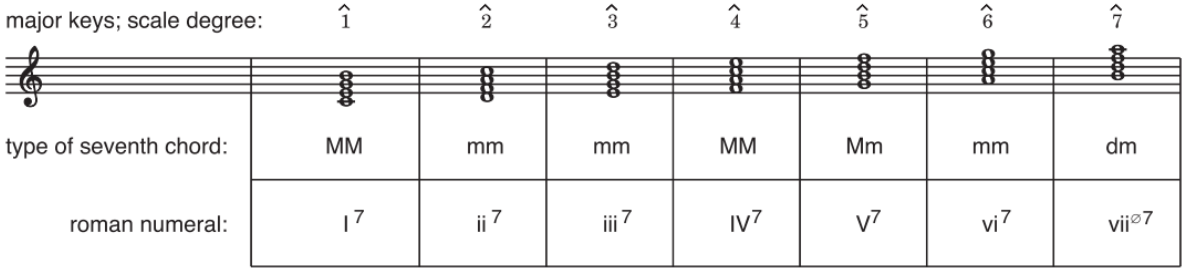

Figure 1. A diagram from The Complete Musician, a music theory textbook, showing Roman numeral annotation of diatonic seventh chords built upon each scale degree C Major. (Laitz, 2012)

Chords are often communicated by musicians via absolute chord names like 'C major', where the only information being transmitted is what notes, or pitches, to play. Absolute chords lack information regarding wider harmonic function. A C Major chord could very well be the **tonic,** or the gravitational centre of the piece, of the key of C major. It could also be the **subdominant** of G Major, or even the

dominant of F major; these terms refer to chords built on the 4th and 5th **scale degrees,** the numbering of the seven pitch classes in the **diatonic** scale (Laitz, 2012), which each possess their own unique harmonic qualities in a piece of music. Although the chord sounds the same in the above three examples, it plays a different role within each key; this is the foundation of what is known as functional harmony. Roman numerals provide contextual information that absolute labels cannot.

RN analysis is founded on the premise that musical chords can be symbolized and labelled according to one of their notes: the **root**. In tertian harmony, chords are comprised of multiple third **intervals** layered on top of a lower, foundational note. For example, a C major **triad** in the root position contains the notes C E and G, each note being three letter names above the other. If the notes are rearranged in a different order, G C E, for example, the chord is in a different **inversion**, but is still a C chord. The lowest note, G, is what is known as the bass note, and C is the root. No matter how the notes are rearranged, the root is still C. In Western music theory, Roman numerals can be utilised to indicate the **degrees of a scale**, such as C Major, which consists of the notes C D E F G A B in that order. Furthermore, if chords are built upon the degrees using every other degree as a chord member as seen Figure 1, Roman numerals in lower and uppercase variations can convey the chord's quality. The case of a numeral refers to the **quality** of the chord's third, which in turn determines if the chord is **major** or **minor**. If the numeral is accompanied by a small plus sign or a circle, then the chord is **augmented** or **diminished**, respectively, though these two qualities are used much less often than major or minor.

For example, the IV chord in Figure 1 is major. In any **diatonic** major key, or a collection of seven pitches in a scale with each letter name represented once (Laitz, 2012), the chord built upon the fourth scale degree will always be major. Instead of reading a chord chart with a song or piece's absolute chord names like "G major" or "F# Diminished", Roman numerals convey not only what notes to play, but the chords' functional relationships within the musical piece itself. Consequently, it benefits musicians and analysts everywhere to utilise and share music annotated with Roman numerals. Only recently has there been a concerted effort to annotate sheet music with RN analyses automatically via machine learning.

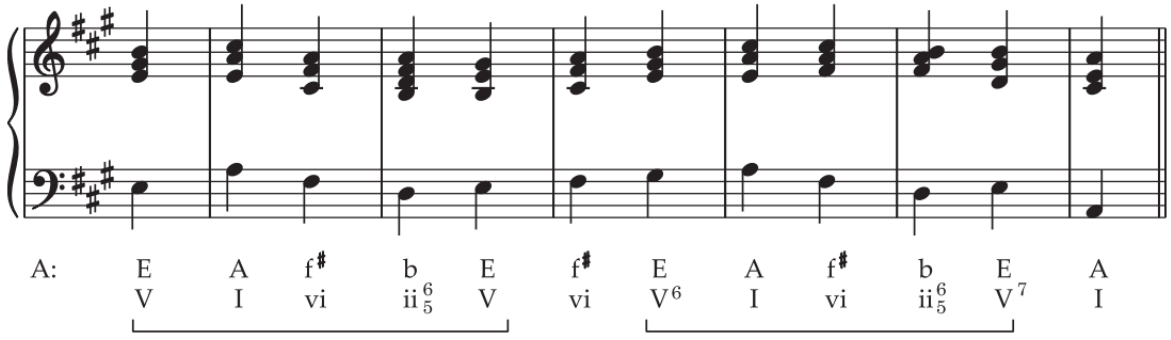

Figure 2. Chord progression reduction of an excerpt from Mozart's "Madaminia" from Don Giovanni as shown in The Complete Musician. This analysis contains both the Roman numeral analysis and absolute chord labels above them. (Laitz, 2012)

Figure 2 illustrates an example of a harmonic **reduction** annotated with both Roman numerals and their inversions, as well as the corresponding chord labels within the diatonic key of A major. The letter names for these chords refer to the chords' root notes, rather than which note is the lowest. It is worth noting, that if a musician knows the current key, and knows the Roman numeral, they can then derive what the root of the chord is. As such, it is not strictly necessary for a model to learn to predict the root, only the key and the scale degree. This project will train the model to predict the chords' roots independently, and then compare the root derived by the key and scale degree predictions as previous researchers have done the same.

1.3 Automated Harmonic Analysis

Many of the previous works on the topic of computational harmonic analysis focus primarily on chord recognition, key detection, and chord sequence modelling. These all fall within the larger functional harmony task, which includes other facets, e.g., chord inversions, key **modulation**, and chord quality. (Chen & Su, 2018). With the rise of deep neural networks, the music information retrieval (MIR) community has achieved substantive progress in key-finding (Nápoles López et al., 2019), lead sheet generation (H.-M. Liu & Yang, 2018), style classification (Choi et al., 2020), and RN analysis. (Micchi et al., 2020)

The data available for the above tasks possess certain caveats that requires clarification. The size and quality of the MIR community's publicly available datasets pale in comparison to those of CIFAR, ImageNet, etc. On top of this, functional harmony annotation is inherently subjective; the ground-truth data can be labelled in several different ways, e.g., two chords can be labelled differently, but when played on a piano, sound exactly the same. In other words, there can be more than one correct chord annotation. These facets make the RNC task an atypical machine learning problem.

1.4 Objectives and Beneficiaries

The intended beneficiaries of this project are musicians, musicologists, and members of the MIR community at large. This work seeks to investigate and apply methods used in other areas of the machine learning community to functional harmony analysis tasks. Those who developed and use these methods stand to benefit from the findings of this report. The main objective of this work is to answer the following research question:

1. **To what extent can components from neural network architectures such as the Transformer** (Vaswani et al., 2017) **and other recently introduced machine learning methods be applied to automatic harmonic analysis in the form of Roman numeral classification?**

The above question can be answered through the design, experimentation, and assessment of neural network architectures using combinations of components, such as multi-headed self-attention, from state-of-the-art models. This project will build on Micchi et al.'s work published in their 2020 paper,

Not All Roads Lead to Rome. We utilise their encoding and pre-processing code to aid in the model development process and ensure that our results are comparable to Micchi et al. (2020). The objectives for the project are elucidated as follows:

1. Design and optimise a bespoke automatic functional harmony model using novel components from the Transformer (Vaswani et al., 2017).

2. Provide an assessment on the performance of the developed model and how it compares to Micchi et al.'s performance for the same dataset.

3. Evaluate and measure the systems' abilities to answer this project's research question.

It should be noted that the project proposal contained an additional research question regarding the alignment of existing audio datasets with their corresponding digital piano rolls for additional feature exploration; this task proved to be too ambitious in conjunction with the first question. As such, the second was discarded from the project scope after conferring with academic advisors who confirmed they found the original goals of the proposal ambitious.

The product of this work is a trained model alongside a new branch of Micchi et al.'s functional harmony Git repository, which includes a very bespoke and open-source pre-processing module that will be utilised greatly in this project. The models and source code developed by the author will be added and clearly distinguished from Micchi et al.'s work.

While this project focuses on the investigation of mechanisms like self-attention as they relate to musical analysis, the learnings of this work possess the potential to inform future applications of language models on non-textual sequence data.

1.5 Work Plan

The work plan roughly followed the stages planned in the project proposal document. A deadline extension was granted, but the phases and their order remained the same.

- **1^{s} Milestone**: Literature review that included evaluation of potential datasets and the reproduction of Micchi et al'.s models and results
- **2nd Milestone**: Investigative research of the encoder stack with multi-headed attention and how others have applied it outside of language modelling
- **3rd Milestone**: Development, optimisation, and evaluation of encoder-based RNC models
- **4th Milestone**: Construct a report summarizing the research process and findings

1.6 Structure of the Report

1.0 Introduction and Objectives, the section above, introduced the concept of Roman numeral analysis, functional harmony, as well as stating the objectives, beneficiaries, and planned milestones of the project. Each of the model's output tasks are introduced and summarised.

2.0 Critical Context provides background on bidirectional recurrent neural networks and the different components of the Transformer architecture that are applicable to this project's task. Each facet contains limitations and advantages that will be explored. This chapter surveys relevant functional harmony literature and how previous contributions advanced the AHA problem set.

3.0 Methods introduces the project's dataset and investigates its properties; the musical relationships visible in the data's distributions are also examined. The input encoding process is then examined, explaining the data augmentation process via transposition used in the original paper. The models' experimental settings and evaluation metrics are defined in this section.

4.0 Results summarises the performance of the various architecture combinations and analyses the hyperparameter relationships observed during optimization experiments. The performance metrics are compared with existing results on the same datasets, and common classification errors between the tasks are examined.

5.0 Discussion critiques and comments on the initial hypothesis, and with the achieved results, reasons as to why said results were observed. The section primarily serves to encapsulate the comparison of the project's outcomes versus the planned objectives and remark on the research process. The model combinations are re-examined analytically, and recommendations made for an optimal configuration.

6.0 Evaluation, Reflections, and Conclusion summarizes and further reflects on the outcomes of this project. Limitations of the results are outlined, and the potential future research areas are identified for those wishing to further this work. The section closes with a perspective from the author on the personal project experience.

2. Critical Context

2.1 Overview

This section provides a general introduction to the recurrent neural networks used by Micchi et al., the Transformer (Vaswani et al., 2017), and breaks down its components applicable to RNC. The concept of relative positional encoding is summarised, and its potential benefits explored. Most of the descriptions of these architectures are drawn from the original project proposal. The multi-task RN problem is introduced, and its components explained.

2.2 Convolutional Neural Networks

Convolutional neural networks (CNNs) are most-used in the task of image classification. They slide a small window across an image, and in doing so, the convolutional layers can learn internal representations of the two-dimensional image and use them to make predictions. These same operations can be applied to a one-dimensional series like sensor data, financial time series, or a sequence of chords and notes in a piece of music. 1-D CNNs do have an advantage over their 2-D counterparts in that the mathematical operations and subsequent sub-sampling operations are on arrays as opposed to n-dimensional matrices. Consequently, the feature extraction and classification operations can be combined in to one parallelizable process due to its relatively low computational complexity. (Kiranyaz et al., 2020) CNNs are an integral part of the existing RNC architectures and are used in Micchi et al.'s convolutional GRU models that employ the use of DenseNet. (Huang et al., 2016)

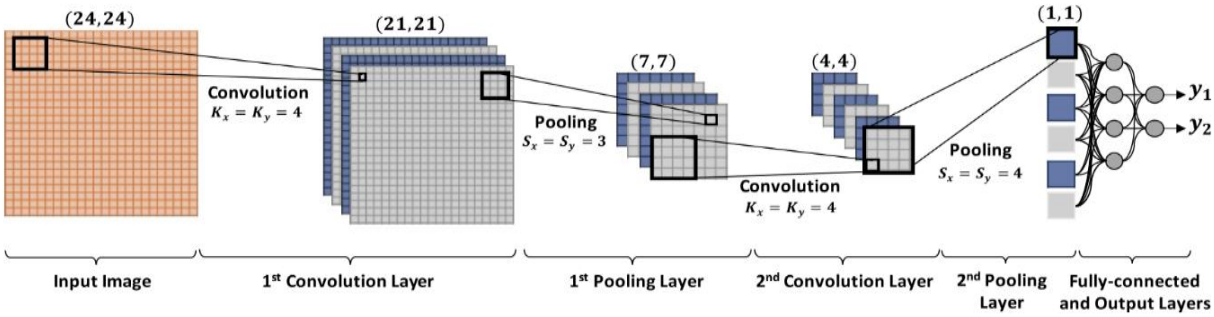

Figure 3. An illustration of a 2-D CNN with 2 convolutional and one fully connected layers (Kiranyaz et al., 2020)

2.3 Bidirectional Recurrent Neural Networks

If a machine learning model is to understand a sequence of data as opposed to an isolated sample, traditional deep networks of multilayer perceptrons (MLP) will not suffice. Recurrent neural networks, (RNNs) can intrinsically associate data with time. Traditional artificial neural networks, like a stack of MLPs, have pre-defined architectures with fixed input and output sizes. These model types do not work well with sequential data of varying length**,** like piano rolls. RNNs maintain an internal state, often referred to as memory, that allows the network to handle and act upon sequences of data of variable

dimensions. Schuster and Paliwal took RNNs a step further when they found that standard RNNs possess restrictions regarding their inability access future input information from the network's present state. (Schuster & Paliwal, 1997) To solve this, they introduced an extension of RNNs coined the Bidirectional Recurrent Neural Network (BRNN); by splitting the state of neurons in to separate parts, one for forward states and one for backward states, the BRNN could be trained without the limitation of using input data up to a pre-set future time step. Micchi et al. used BRNNs in their work on Roman numeral analysis and set the baseline this project will attempt to improve upon.

2.4 Baseline models

The basis for results comparison will be those of Micchi et al. produced using their convolutional GRU models comprised of many stacked Dense Net layers (Huang et al., 2016). Their model architecture first convolves in 1-D along the time domain, encoding feature maps of chord time-steps in different channels, similar to colour channels in image analysis. (Micchi et al., 2020) The bidirectional GRU block then uses information from both past and future frames to allow for the discovery of complex correlations in the input sequences. Their architecture concludes with a block of fully connected layers with softmax activation functions for each RN component task.

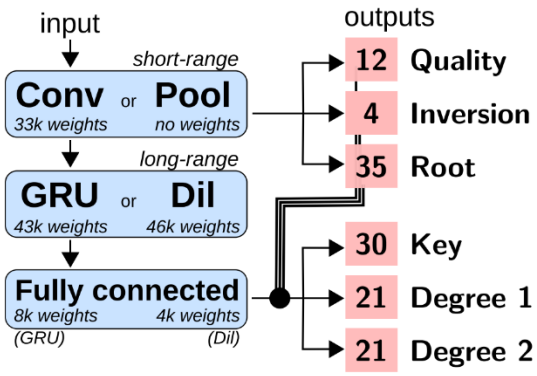

Figure 4. Diagram of Micchi et al's functional harmony classification model. The numbers refer to the number of categories for each output label, the same as used in this project's model.

Figure 4 contains a visualisation of their model architectures; one important distinction is that their model optionally omits three tasks (*Quality*, *Inversion*, and *Root*) from the GRU block. It was hypothesized by the researchers that certain tasks benefitted from short-range feature-extraction in comparison to others that would benefit from computations, e.g., the GRU block, that made use of information from a larger chord context of 40 eighth notes in forward and backward directions.

Micchi et al achieved 42.8% overall RN accuracy on their validation dataset. This metric, along with the accuracies of each individual RN task, will be the baseline for comparison and the evaluation of this project's outcome.

2.5 The Transformer

Vaswani et al. introduced the Transformer architecture in 2017 in their paper *Attention is All You Need*. This model, unlike other dominant sequence transduction models, does not utilise recurrence or convolutions, and instead utilises what is known as self-attention. In previous state of the art sequence models that made use of convolutional neural networks, the number of operations required to relate signals from two arbitrary input or output positions grows in the distance between positions (Vaswani et al., 2017), making it computationally inefficient for sequences of longer lengths like music. The Transformer reduces this to a constant number of computations.

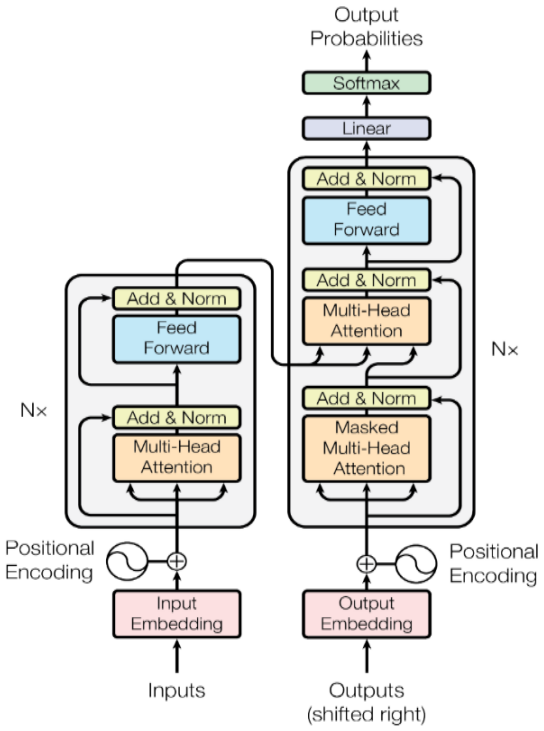

Figure 5. The Transformer model architecture introduced in Attention is All You Need (Vaswani et al., 2017)

The Transformer is comprised of an encoding component and a decoding component. The former is a stack of N encoder layers, each containing a multi-headed self-attention layer, which will be discussed in its own section, and a feed forward neural network; the decoding layers are the same, except for the addition of an additional attention layer to identify relevant portions of the input sequence.

The Transformer was initially developed for sequence-to-sequence tasks like machine translation, e.g., translating a sentence from French to English. However, components from the architecture have found success outside of NLP in areas such as time series forecasting (N. Wu et al., 2020) and video frame synthesis. (Z. Liu et al., 2020) The encoder on its own can learn relevant portions of an input sequence and serve as a pre-processing step prior to a classification layer in lieu of a decoder. As such, in this work, the encoder/decoder structure was not directly applicable, and only the encoder portion (left side) as seen in Figure 5 was investigated.

2.5.1 Multi-Headed Self-Attention

Vaswani et al.'s most notable novel contribution is the self-attention mechanism. As their model processes the tokens of the input sequence, self-attention lets the model see the other words in the sequence for learnable properties that it can attribute to a more relevant encoding. In other words, the self-attention layer allows the encoder to inspect the other data in the input sequence while it's encoding

a specific token or data point in that same sequence. Recurrent neural networks maintain a hidden state that serves a similar purpose, managing representations of the previously seen tokens.

Self-attention is calculated by creating three vectors from the input; in typical language tasks this input is a word embedding, a semantic representation of the word in vector space. (Mikolov et al., 2013) For RN analysis, the input vector is a piano roll with pitch information as features. The three created vectors are named Query, Key, and Value. These are created by multiplying each input vector by three matrices, $\mathbf{W^Q}$, $\mathbf{W^K}$, and $\mathbf{W^V}$ that are learned during training. The **Q**, **K**, and **V** vectors are abstractions that facilitate the creation of an attention score, calculated as the dot product of **Q** and **V** for each token. For example, when calculating a self-attention for a chord at index 0, the first score is the dot product of Q_0 and K_0, and the second score is the dot product of Q_0 and K_1. This product is calculated for each index of K, which allows the model to learn the relevance of other tokens relative to the current index.

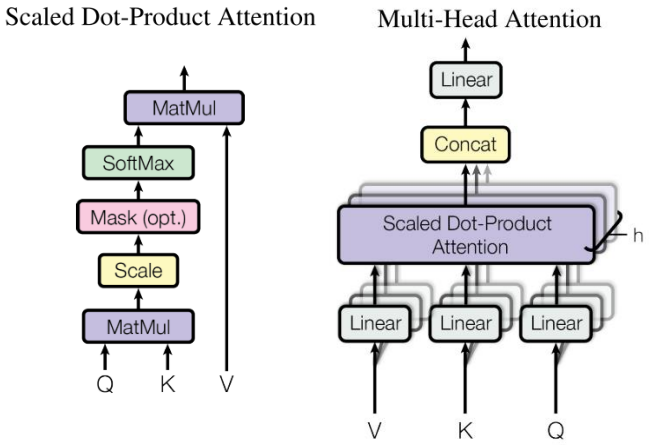

Figure 6. The multi-head attention mechanism as illustrated in Attention is All You Need (Vaswani et al., 2017)

As the name implies, the attention block repeats the above calculations several times in parallel, where each of these computations is known as an attention head. Within the attention block, the **Q**, **K**, and **V** vectors are each split N different ways. Each split vector is then processed by its own head. The resulting N matrices with attention scores are concatenated together. This mechanism improves the model's aptitude when focusing on tokens at other indices. In the context of RN analysis, this is the mechanism that should allow the model to learn that a V^7 chord is often followed by a I chord, for example.

In their 2019 paper *Are Sixteen Heads Really Better than One?*, Michel et al. found that multi-headed attention does not always fully leverage its theoretically superior expressiveness over vanilla attention as it relates to encoder-decoder models. In other words, when they conducted experiments pruning the attention mechanisms of models like BERT (Devlin et al., 2018), they found that heads can be removed from trained transformers without degradation in test performance, and that some layers could be reduced to only one head. (Michel et al., 2019) The relationship between the number of attention heads and RNC performance will be explored in this project.

2.5.2 Positional Encoding

Because the input sequence is split many ways, and the model does not contain recurrence or convolution, it cannot internalise the ordering of the full input sequence. This is the caveat of parallel computation as opposed to processing the sequence token-by-token. Similar to an LSTM (Hochreiter & Schmidhuber, 1997), the Transformer model does not know the sequential order of the tokens when the input is fed to different attention heads. To rectify this, a positional encoding vector is added to each input vector. This positional encoding vector is constructed using a specific sinusoidal pattern that is learnable and assists the model in determining the position of each token, or the distance between them. Rhythmic patterns are very strong in music; positional encoding should allow this project's models to learn concepts like **syncopation**, and how changes in rhythm inform harmonic progressions.

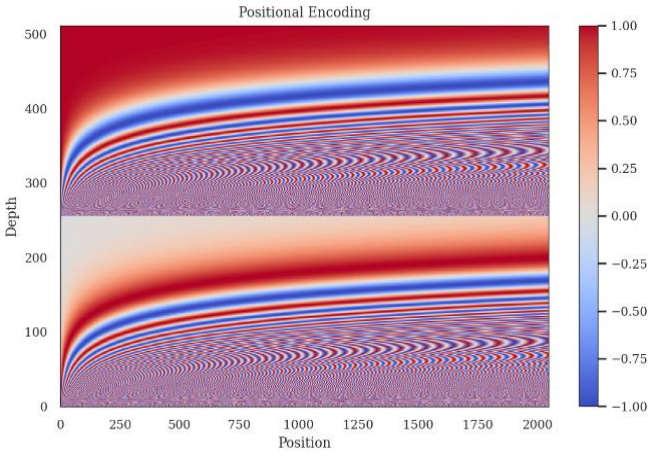

Figure 7. Positional encoding with 512 dimensions and a maximum length of 2048

2.6 Relative Positional Encoding

Absolute methods of encoding positional representations, including the application summarised above, encode the positions of input tokens deterministically from 1 to the maximum sequence length. This means that each position has an individual encoding vector. (K. Wu et al., 2021)

Something missing from the Transformer architecture is the ability to capture relative positions and inform the model on tokens' relevance, something that convolutional neural networks capture within the kernel size of each convolution. (K. Wu et al., 2021) Shaw et al. from Google Brain introduced the concept of self-attention with relative position encoding representations in 2018. Their work extended the self-attention mechanism to efficiently consider encodings of relative positions, or distances between sequence elements. (Shaw et al., 2018)

The researchers at Google thought that for certain tasks, the distance between tokens is sometimes more important than their absolute positions. On top of this, the Transformer as introduced in 2017 could only take sequences with a length up to a certain maximum. Shaw et al. realized this and developed a new method, titled Self Attention with Relative Position Representations, that clips the maximum distance to a parameter k on either side of the token. Because their method is bidirectional, $2k + 1$ elements are considered.

Because this project is a multi-task problem, where each of the tasks including *Degree*, *Quality*, *Key*, and *Inversion* may benefit from focusing on different parts of the input sequence to try and capture local patterns. A potential application could be limiting the scale degree encoder to only look at one or two measures ahead/behind the current token, because the immediate chords preceding it inform the scale degree more than a chord in a previous phrase. Because a chord's inversion is based solely on the ordering of the chord from the lowest to highest note, and not sequentially dependent, accuracy may increase if the encoder for inversion is restricted to certain ranges before and after the current vector. This would be because irrelevant information, information that's further away, is being discarded or penalized.

2.7 Multi-Task Learning

Roman numeral chord labels consist of 4 components: *Degree* (includes Degree 1 and Degree 2), *Quality*, *Key*, and *Inversion*. An example of a RomanText entry, the format of this project's dataset labels, for a measure could appear as follows: `m29 b1 I64 b2 V7/V`. In measure 29, an inverted tonic triad in a major key is followed by a root inversion **secondary dominant** chord in the same major key, indicated by the presence of a backslash in the second chord annotation. Secondary dominants are summarised in a future section. Each of the 4 components of a RN label should be treated as their own problem with models and parameters tweaked to its specific behaviour. A generic formula for a complete Roman Numeral label is constituted below:

	RomanText	Start	End	Key	Degree	Quality	Inv.
m1	c: i	0	4	c	1	m	0
m2	iv64	4	8	c	4	m	2
m3	viio643	8	12	c	7	d7	2
m4	i	12	16	c	1	m	0
m5	bVI6	16	20	c	-6	M	1
m6	V642/v	20	24	c	5/5	D7	3
m7	v6	24	28	c	5	m	1
m8	V642/iv	28	32	c	5/4	D7	3

Figure 8. The RomanText RN annotations for the first 8 measures of Bach's 2nd prelude, aligned with chord properties

RN = Degree 1 & 2 + Key + Quality + Inversion

In the above example annotation, the first chord would be broken down as follows:

I64 = I (1st) & I (2nd) + Major (mode) + Major Triad (quality) + 2nd Inversion (inv)

Some of these RN elements are informed by one another; in other words, they are not all independent. Importantly, each of the constituent tasks must be correct for the record to be considered correctly annotated by the model. Each of these RN components and their relationships to other tasks are further explained below. Music theory concepts are introduced throughout to provide context to the types of harmonic properties and relationships that the model could learn for each task.

2.7.1 Degree Task

The scale degrees of the two RN chord labels in the example `m29 b1 I64 b2 V7/V` are the I and V from I64 and V7/V respectively. The case, or quality of the degree i.e., major or minor, is not a component of this task; only the numeric value of the scale degree is what the model will predict. Values range from 1, the tonic, to 7 the **leading tone**, with potential for accidental modifications.

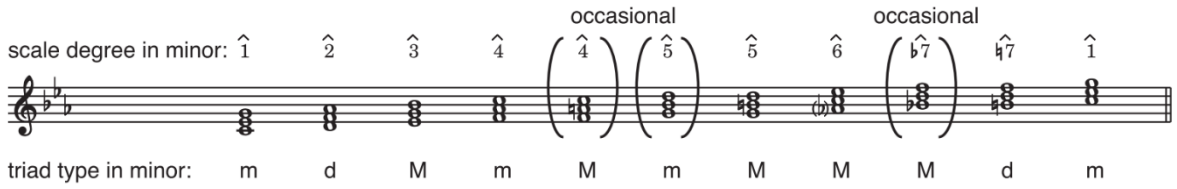

Figure 9. Scale degrees and triad types in a minor key as seen in The Complete Musician. Scale degrees are indicated by the carat symbol above the number. The letters below the staff indicate the chords' qualities, which are explained in another section. (Laitz, 2012)

Figure 9 displays the scale degrees of the key of c minor, as well as the triad qualities built off of the c minor **diatonic** scale. Scale degree is a multi-task problem in itself. The secondary entry of the RomanText annotation example from above is what is known as a **secondary dominant** chord, a V7/V, which indicates the use of non-diatonic pitches from an adjacent key. In this specific instance, the relation is the V7 chord of the current key's dominant (V). Usages of non-diatonic chords are relatively rare, but still need to be accounted for in the dataset and prediction process. Most 1st degree labels (denominator in the above example) are I because they are in the current key, not using tones borrowed from another.

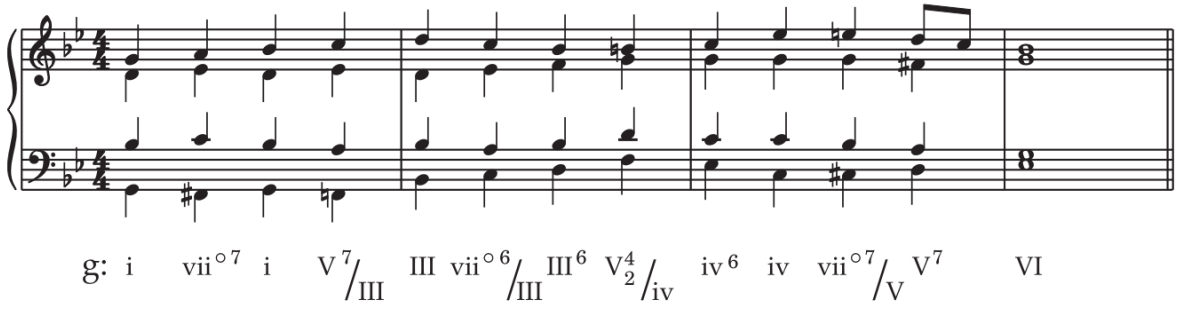

Figure 10. An example of secondary dominant usage in g minor from The Complete Musician. (Laitz, 2012)

An example of secondary dominant usage can be seen in Figure 10. On the 4th beat of the 2nd measure, a B natural note is introduced, indicated by the ♮ symbol to the left of the note, which is not found in the key of g minor. This B natural turns what would be a G minor chord in to a major one, and in turn creates a feeling of tension that yearns for resolution to a chord other than the original key's tonic. In this case, this would be the iv^6, which in the context of g minor, is a c minor chord. In this example, the *Degree 1* of the fourth chord in measure two would be iv, and not i, because it's resolving to a *secondary* key. This is an example of a modulation, or a small departure from the tonicized key. For the RN labels

that do not have a fractional annotation, it is implied that the *Degree 1* of this RN for this project's usage is the tonic of the current key.

This added nuance of borrowed harmony makes predicting *Degree* require that secondary dominants are correctly identified in addition to the scale degree upon which the chord is built. This may be difficult for this project's model to learn due to how sparsely these types of chords occur in the dataset.

2.7.2 Key Task

The key specifies the mode and pitch spelling of the primary diatonic scale in use. The pitch spelling refers to the set of pitches comprising the diatonic scale, and their **accidentals** (sharps and flats). The G♭ major scale is *G♭, A♭, B♭, C♭, D♭, E♭, F*; these notes, when played on the piano, vibrate at the same exact frequencies as *F♯, G♯, A♯, B, C♯, D♯, E♯*. These pitches are known as **enharmonic** equivalents. The model should learn how to recognize whether a piece is in *G*♭ major as opposed to F# major, even though they sound identical. Additionally, the **mode** of the key needs to be learned. This is whether the key is major or minor, indicated by the case of the key label, e.g., *e♭* corresponds to E flat minor when *E♭* refers to E flat major.

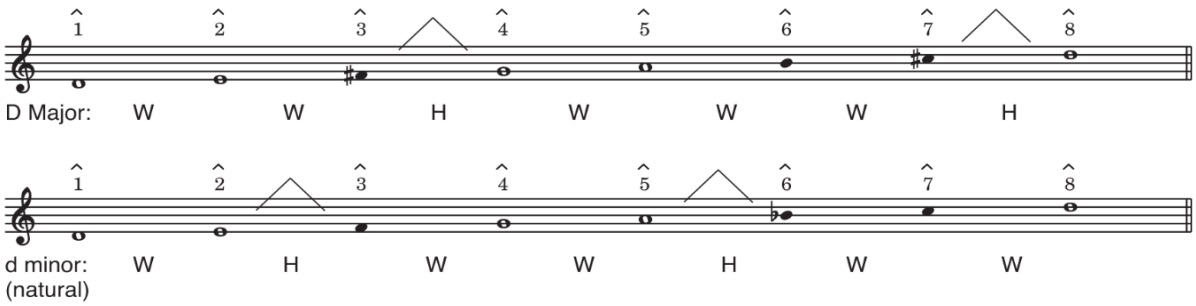

Figure 11. A comparison of the D Major and minor scales as depicted in The Complete Musician. Letters W and H refer to the whole and half-step intervals between certain scale degrees, respectively. (Laitz, 2012)

Figure 11 compares the D major and d **natural minor** scales. While they share the same tonic, the intervals between scale degrees differ, and so do their **accidentals**. A model's task would be to correctly identify the current pitch-set, or the predominant diatonic scale in use, as well as indicating the mode: major or minor. Micchi et al. found the best success using a full pitch spelling approach, which treats enharmonically equal scales differently, thus creating 30 possible target values for this task.

It's hypothesized that the key's mode informs which scale degrees might be used. For example, a iio7 chord is much more likely to occur in a minor key than a major one. When the output of one task, like the key, can inform the outputs of other tasks, like the scale degree, there is opportunity to chain model architectures together to promote RNC performance via inter-task relationships.

2.7.3 Quality Task

A chord's quality is defined by the properties of the component intervals within the chord. For example, in the following chord annotation, `m26 i b4 iio7`, the quality of the first entry is a minor triad, which does not contain a seventh, indicated by the individual numeral's lower case. The chord on beat 4 of the measure in this example is a fully diminished seventh chord, indicated by the `o7` next to the degree. The circle symbol corresponds to being fully diminished. Figure 12 illustrates how a diminished triad contains two minor 3rd intervals (3 semitones) on top of one another, and how a minor triad differs by only one note, containing a **perfect fifth** instead of a **diminished** one. Chords with diminished triads are used less frequently than major and minor chords.

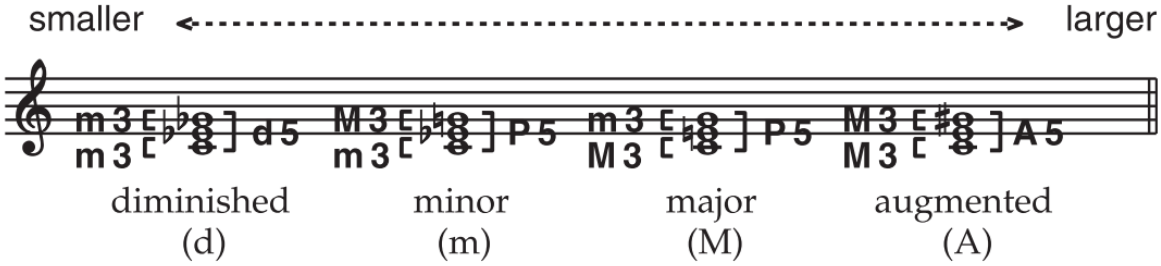

Figure 12. An example of triad types and their constituent intervals from Laitz's The Complete Musician (Laitz, 2012)

A machine learning model should be able to distinguish the underlying structures of these different chord qualities and classify them accordingly. As with the other tasks that contribute to a full RN classification, a multi-class classifier will be required, there are 12 possible qualities as identified by Micchi et al., *M, m, d, a, M7, m7, D7, d7, h7, Gr+6, It+6,* and *Fr+6*. The last three are augmented 6th chords that are relatively rare but are still considered for classification.

2.7.4 Inversion Task

Chords with notes permuted into different orders still serve the same harmonic function and retain the same root; the ordering of notes in a chord can be referred to as its inversion. When the lowest note of the chord is not the same as the letter it's named after, the chord is inverted. It is worth denoting that the root of a chord is the pitch upon which a triad is built, and the bass note is the lowest one, which

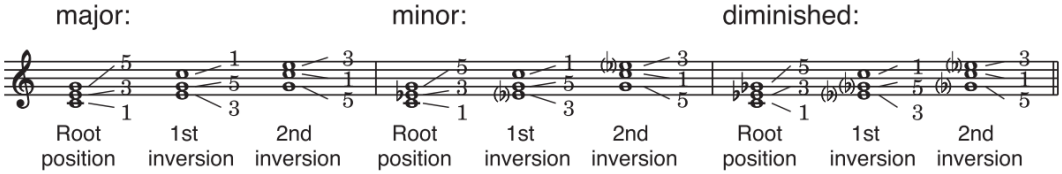

Figure 13. Triad types and inversions as exemplified in The Complete Musician (Laitz, 2012)

when in root position, are one and the same. However, when a chord is inverted, the 3rd, 5th, or 7th (when present) pitches can be in the bass. Figure 13 showcases the differing triad inversions and how they appear in musical notation. It should be noted that seventh chords have 4 possible inversions while triads only have 3. There are 4 possible inversion classes from 0 to 3 in the dataset as target values for

the model. Figure 14 shows a similar diagram as above for seventh chord inversions, but this time using **figured bass** notation to convey the inversion ordering, e.g., 6/5 or 4/3. Figured bass conveys the distance between each chord member and its distance to the bass note. Notice that there are 3 possible chord orientations other than the root position. Seventh chords occur less frequently than triads in this project's dataset, which introduces class imbalance for the *Inversion* task. It is hypothesised that a model will experience difficulty predicting seventh chord inversions as they are not as prominent in the dataset.

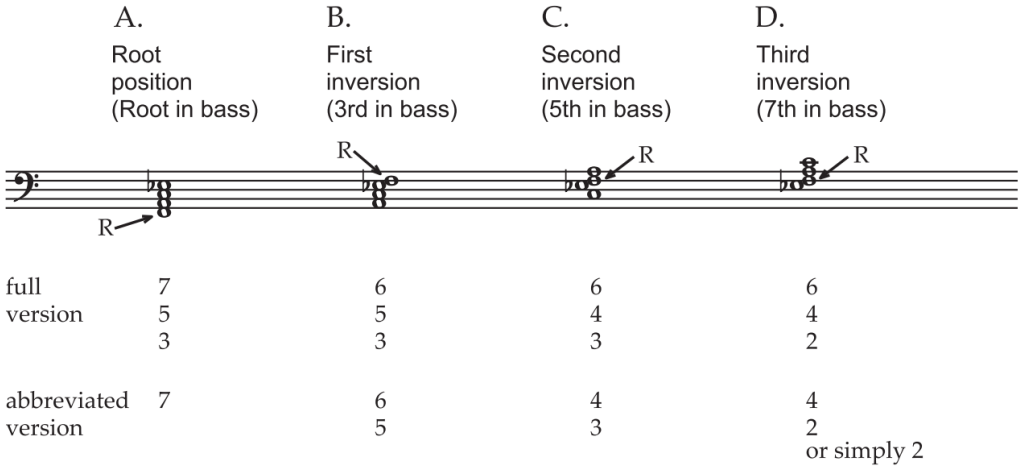

Figure 14. How seventh chord inversions are defined as seen in The Complete Musician using figured bass notation. (Laitz, 2012)

2.8 Conclusion

This section contained the summaries of relevant neural network architectures used by Micchi et al. to achieve the results that will be the benchmark for this project. It walked through the elements of the Transformer architecture and the potential applications of its encoder stack with multi-headed self-attention. The concept of positional encoding was introduced, and the potential benefits of relative positional encodings were propositioned. It contained a description of the multi-class task to be learned by an ensemble of models, where each sub-task is explained more in depth alongside music theory concepts, and the idea of beneficial inter-task relationships was explained.

3. Methods

3.1 Overview

The methods section serves to illustrate the processes taken to answer the research question and achieve the objectives of this project; it contains the steps taken to assess and better understand the feasibility of components like the self-attention mechanism and relative positional encoding as it pertains to RN automatic harmonic analysis.

First, the details of the project's datasets are explored, and the subtleties of each constituent dataset are summarised. Properties and statistics of the dataset are computed and commented upon, including distribution of tonics, RN degrees, and other chord-related properties. The input piano roll encoding process is explained, with the importance of bass notes at each time-step elucidated for the purposes of inversion classification. The data augmentation process via transposition is summarised, and its motivations affirmed. The project's multi-task model architecture is then visualized and the architecture/hyperparameter experimentation process is described. Two learning rate scheduling methods are expounded upon and several of the project's evaluation metrics are defined.

3.2 Research and Requirements

A survey of MIR literature, specifically those focusing on functional harmony classification was conducted to not only inform modelling decisions and architectures, but also to assess and avoid pitfalls encountered in previous research. Additionally, literature concerning the Transformer architecture and its derivatives was examined. Transformer-based models, unlike the convolutional GRU architecture used by Micchi et al. with less than 100,000 learnable parameters, require millions. For example, Google's BERT (Bidirectional Encoder Representations from Transformers) model family introduced in 2018 contains 340M parameters. (Devlin et al., 2018) These large models are best trained with a dedicated GPU leveraging CUDA, a toolkit from NVIDIA that provides GPU-accelerated model optimisation. A CUDA compatible NVIDIA RTX 2080 Super with 8GB of VRAM was used during this project's experimentation and model refinement process.

3.3 Datasets for Developing Experimental Architecture

Several datasets of human harmonic analyses have been published since 2010, all including RN annotations on Western Classical pieces. Among them, the TAVERN dataset from Ohio State University, an acronym for Theme And Variation Encodings with Roman Numerals, consists of "27 complete sets of theme and variation for piano" from Mozart and Beethoven. (Devaney et al., 2015) Each of these pieces were annotated by PhD music theory students at Ohio University; harmonic annotations are not something easily crowdsourced like other types of datasets are. Two different

annotators analysed each theme and variation to ensure consistency, sometimes disagreeing. The TAVERN dataset is the oldest sub-dataset investigated and utilized during this work.

The Annotated Beethoven Corpus, ABC, is another labelled corpus containing harmonic analyses of all sixteen Beethoven string quartets, encoded in MuseScore XML format. The open-source format allows humans to interpret the data intuitively while also being machine-readable. (Neuwirth et al., 2018) This improvement over the more complicated **hern format allowed analysts to annotate and create the truth data while listening to the music; ABC contains the harmonic analyses of all seventy movements of Beethoven's 16 string quartets.

```
Composer: J.S. Bach
Title: Prelude No.20
Analyst: Mark Gotham

Time signature: 9/8

m1 b1 a: i b2 i b3 i6
m2 b1 ii/o42 b2 viio b3 ii/o65
m3 b1 iv764 b2 viio7 b3 iv543
m4 b1 iv4 b2 i64 b3 e: viio43
m5 b1 i b2 i64 b3 i
m6 b1 ii/o7 b2 V65 b3 ii/o7532
m7 b1 viio643 b2 viio653 b3 V6432
m8 b1 iv43 b2 i64 b3 viio5/v
m9 b1 i b2 i64 b3 d: V7653
m10 b1 i b2 i64 b3 C: V7653
m11 b1 I b2 I64 b3 IV763
m12 b1 V b2 V65 b3 iii6
m13 b1 I b2 vi7542 b3 I
m14 b1 IV64 b2 IV64 b3 IV64
m15 b1 viio742 b2 V6 b3 V43
m16 b1 I b2 I64 b3 I
m17 b1 g: viio7 b2 V65 b3 vii
m18 b1 i b2 i64 b3 i
m19 b1 d: viio7 b2 V65 b3 viio
m20 b1 i b2 i64 b3 i
m21 b1 a: viio43 b2 viio6553 b3 V6432
m22 b1 i6 b2 i b3 i6
m23 b1 bVI b2 i b3 i
m24 b1 iv b2 ii b3 viio
m25 b1 i6 b2 bVI64 b3 iv5 b3.67 V3
m26 b1 I b2 V7/IV b3 I753b2
m27 b1 iv654 b2 ii/o42 b3 viio643
m28 b1 I
```

Figure 15. Example of the RomanText standard for J.S. Bach's Prelude No. 20 in A minor, BWV 865

Chen & Su contributed the Beethoven Piano Sonata with Function Harmony, BPS-FH, in 2018, which contains the musical data and functional harmony annotations of the 1st movements of 23 of Beethoven's Piano Sonatas (Chen & Su, 2018). BPS-FH was the first RN annotated corpus with harmonic analyses on a step-by-step resolution, where previous datasets are primarily chord-change based. Researchers annotated 5 chord functions: key, primary degree, secondary degree, quality, and inversion. Because sonatas are prone to modulation, Chen & Su recorded changes of keys by specifying a local key, allowing to record how the temporary tonic differs from the global tonic, where the piece started. (Chen & Su, 2018)

The previous datasets were all annotated in different formats, each being relatively small by their own merits. Tymoczko et al "recognised the absence of a large public corpus of expert RN analyses, and a fundamental lack of an agreed-upon, computer-readable syntax in which those analyses might be expressed." (Tymoczko et al., 2019) To remedy this, researchers presented a new combined dataset containing 5 different datasets: Monteverdi's madrigals (48 pieces), Bach chorales (20 sampled

analyses), Bach's Well Tempered Clavier preludes (24 pieces), the ABC corpus (70 pieces), and the Scores of Scores corpus (50 pieces).

With the goal to create a format easily parsable by a computer while remaining easy for humans to read and write, Tymoczko et al proposed a framework that presents data in the following format: one line per measure, with each line of analysis starting with the symbol 'm' to denote the measure, then pairs of beat numbers with their corresponding Roman numerals. As an example, `m6 b2 V6 b3 vi` indicates that in measure 6, a V6 chord is on beat 2, with a vi chord on the third beat.

The format is very succinct and readable while maintaining the ability to describe any standard tonal chord. An example of a J.S. Bach piece transcribed in the RomanText format can be seen in Figure 15. Measure 9 for example, conveys an E minor chord, the Tonic chord in root position due to an earlier key change, followed by an inversion of the same chord, and a key change to d minor using an inverted Dominant 7th chord. This simple yet elegant syntax conveys all the chord and note values, their durations, and maintains the secondary dominant relationships, all in a concise and readable format. Micchi et al. drew from all the above datasets to create their own combined dataset utilizing TAVERN, ABC, BPS-FH, and RomanText, in the RomanText format.

3.3.1 Properties of the Data

Table 1. Various properties of the dataset used by Micchi et al. and in this project

Dataset Property	**Value**
Number of Scores	205
Median Piece Duration (in quarter notes)	471.25
Total Quarter Notes in Dataset	127,287
Median Num Key Changes per Piece	5

With pieces from various composers, Micchi et al's combined dataset contains 205 unique works, as seen in Table 1; each of them with a RomanText file, a scores .xml file, and a .csv file containing chord labels. Classical composers like Beethoven did not shy away from modulations and borrowing chords from second/tertiary dominants; with a median of 5 key changes per work, algorithmic applications like that of deep learning benefit from non-diatonic harmony.

The distribution of keys, the major or minor scale around which a piece of music revolves, can be seen in Figure 16. The most popular keys are not surprising: C major and c minor, followed by F and G, which (except for C minor) contain the fewest sharps and flats on the piano, which may be a factor in their observed prominence. They are also close to one another in the **circle of fifths**; looking at the major keys graph in Figure 16, we can see B♭, E♭, and A♭ in sequence of descending number of observations.

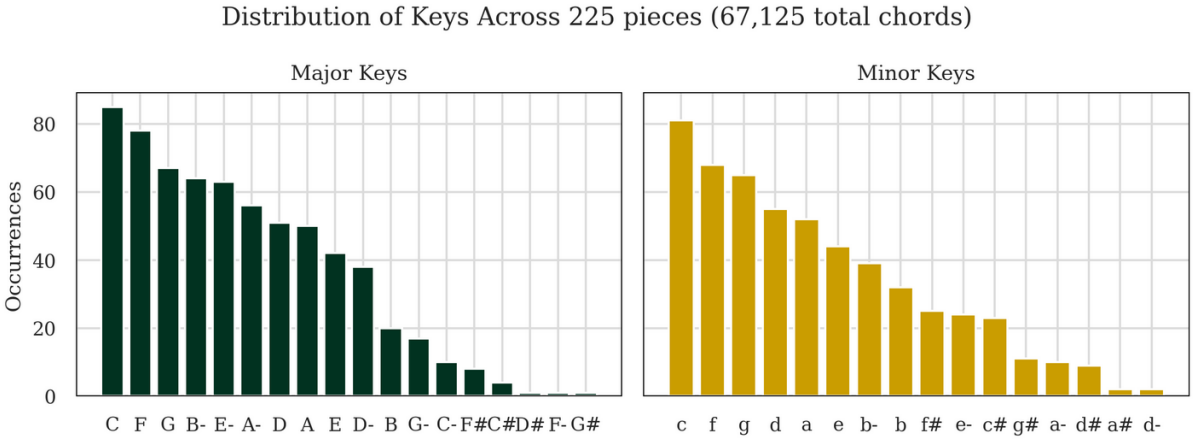

Figure 16. Distribution of Major and Minor tonics (keys) found in the datasets in descending order of prevalence

The **circle of fifths**, a diagram displaying the tonics of the twelve major or minor keys ordered a fifth apart along the periphery of a circle (Rudbäck & Niklas, 2020), is a common way of visualizing how various tonics relate to one-another. The large diagram can be found in the appendix of this report. Music theorists, composers, and musicians will utilize the circle of fifths to discuss and analyse the relationships between keys and the pitches within them. Figure 17 illustrates how pitches in various flat keys are shared by other flat keys. E♭ and B♭ are a perfect fifth apart; they share almost every note with one another, except for one. This means they are sonically similar to one another, and modulations, or changes in key, are not as jarring to the ear.

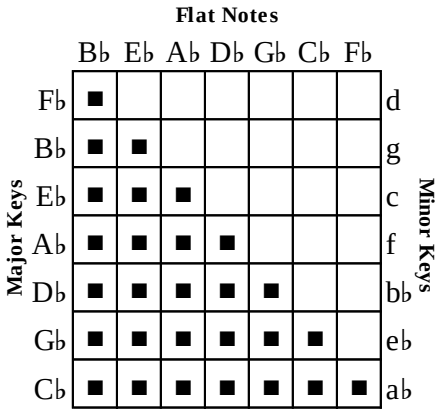

Figure 17. A visual representation of the notes shared between flat keys, as well as illustrating the relative major and minor pairings

This relationship is something that a machine learning model should be able to learn. Part of the classifier's task is to accurately predict when a chord from a relative key is used via a dense neural network. During a piece in B♭ major for example, a II major chord (C major) deviates from the diatonic scale by introducing a an E♮, a note not in the B♭ major scale. It is then much more natural to transition to F major, the tonic chord to the C major chord's dominant. This chord's Roman numeral would be labelled as a V/V, indicating that it is the V chord (C major) of the current key's dominant chord (F major).

The chordal relationship described above is a secondary dominant, a concept introduced earlier. Figure 19 visualises the most frequently observed scale degrees in the dataset. It is observed that the I and V degrees, as hypothesised, occur more frequently than any other. Of the secondary dominants, the V/V, or a ii chord with a raised third, appears most frequently. It is validating to see that the dominant is the most borrowed-from key, because it shares the greatest number of pitches with the original key. The relationships of adjacent keys, and their corresponding similarities, should be something multi-task encoder model can learn.

Chords can be labelled as one of 12 distinct qualities, which are illustrated in Figure 18. Of the 12 possible quality labels, chords labelled M, or major, are the majority at 45.6% of chords. D7 chords, major triads with a dominant 7th, are second-most prominent at 24.1% of all chord occurrences. Of the 4 possible inversions, the way the notes in the chord are ordered, chords in root position are most common. They constitute 58.2% of the inversion observations. This aligns with expectations, as root position chords are common in classical music. It should be noted that 3rd position inversions are only possible for 7th chords, as there are 4 possible pitches as opposed to 3. In other words, a triad, which only has 3 pitches, cannot be ordered in more than 3 ways.

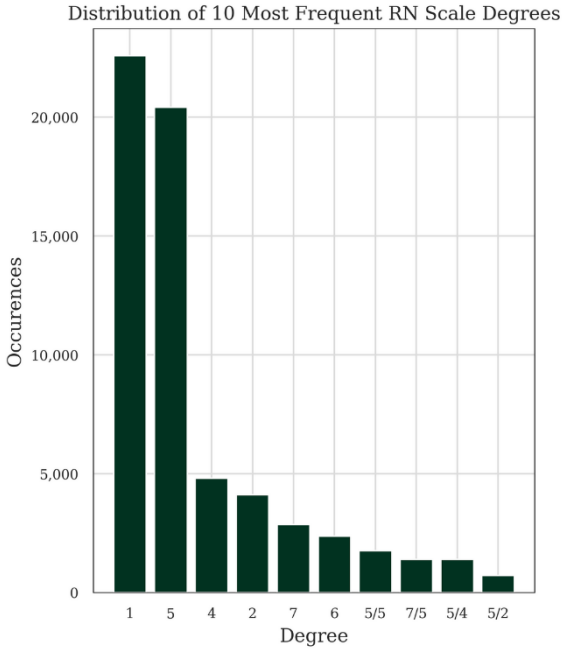

Figure 19. The most frequently occurring scale degrees in the project's dataset

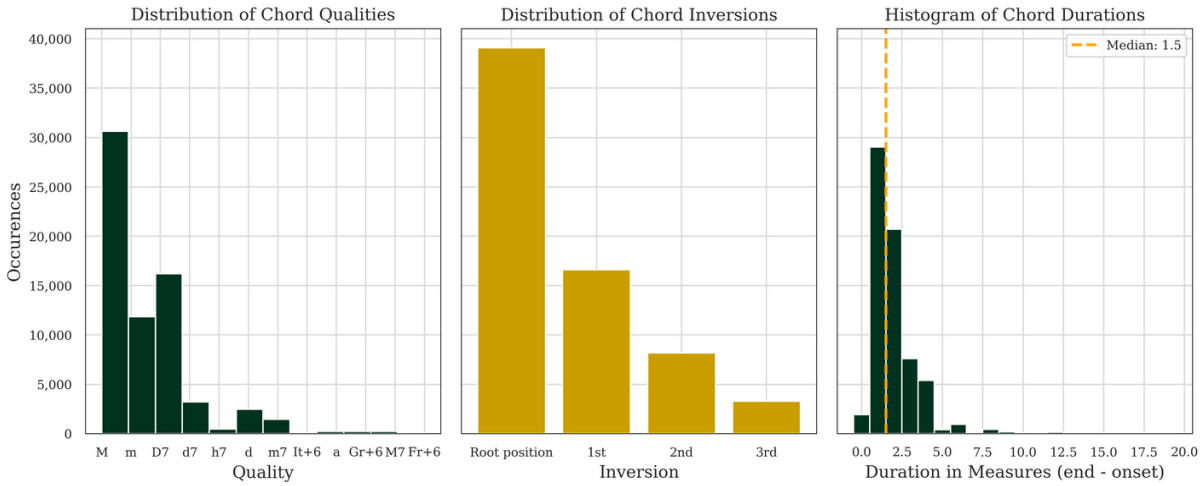

Figure 18. Visualisations displaying the dataset's various chord properties

The right-most chart, a histogram of chord durations, displays how long a chord is held for in measures. On a median basis, chords are held for 1.5 measures without much deviance. With a Fisher's Kurtosis

21.25, an indication if data are light-tailed or heavy-tailed when compared to the normal distribution, durations held for longer than 5 measures should be viewed as anomalies within the context of this task.

3.3.2 Encoding Model Inputs

In the pre-processing step, the score .mxl files, that contain the pitch and **rhythm** values for each piece, are parsed and loaded via music21, an open-source Python library used as a toolkit for computer-aided musicology. (Cuthbert et al., 2010) In addition to note values and durations, Music XML files contain attributes like **time signatures**, note stem directions, **ornaments**, and written repeats. With the aim of creating as robust of a dataset as possible, when a range of measures contains a repeat, they are expanded prior to encoding.

Following Micchi et al., a vectorized frame-based approach is used to represent and encode time. Each input vector represents an individual time-step, where the frame resolution equals a 32nd note; there are 8 chord time-steps per quarter note. Figure 20 displays the input representation matrix for J.S. Bach's *Prelude No 20 in A Minor*. The upper-most band of pitches contains the total pitch content of the frame, with the bass note extracted and concatenated. Adding the bass note informs the system when determining chordal inversions as well as harmonic progressions. (Micchi et al., 2020) The researchers

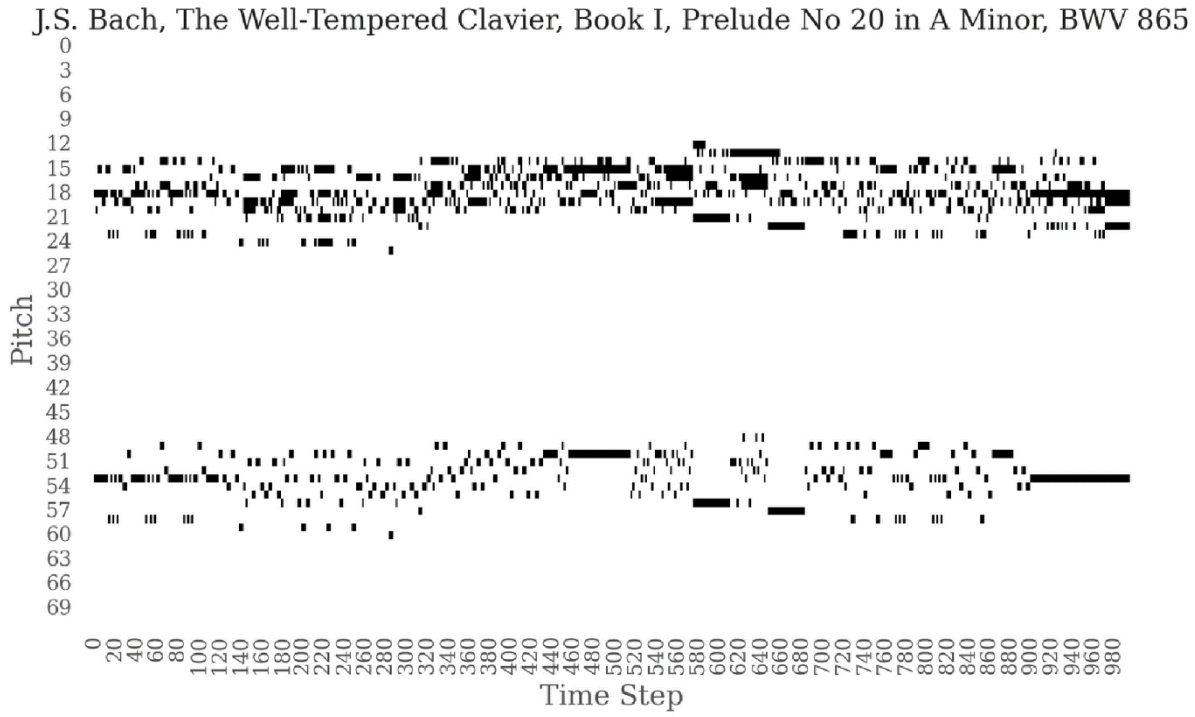

Figure 20. The input vector of a J.S. Bach prelude. Pitches 35 and above encode the bass note of the chord, with the aim to inform the model when classifying inversions.

in the original paper excluded pitches outside of a seven-octave range, as notes outside of this range are rare and likely do not contribute harmonically relevant information. Each piece is loaded into a matrix

like the one displayed, and sliced into smaller sub-matrices of varying sizes, the optimal of which will be determined by experimentation.

The purpose of splitting up the scores into smaller chunks is to assist the model in generalising patterns and to prevent overfitting to individual compositional works. When encoded, the pitch values are associated with their corresponding note names as opposed to a discrete numeric range; this allows the system to learn the nuances of enharmonic equivalents. When parsing the score files, the flattest and sharpest notes in the piece are recorded and stored for future use when determining how many variations of the piece are feasible to create for data augmentation.

3.4 Data Augmentation

In image-based classification tasks it is common to augment the dataset by rotating the input images, introducing gaussian blur, or randomly omitting pixels to promote generalisation. When considering augmentation techniques for piano roll data, Micchi et al. implemented transposition, or key-shifting, where the entirety of a composition's pitch content is translated to a new key and is effectively a different piece. This creates new training samples for the model to learn from. Though, it is practical to recognize that two transpositions of the same score are not as dissimilar as two separate compositions are. To limit the pitch-space, because pieces can be transposed infinitely, pitches are restricted to being labelled F𝄫 through B𝄪. The flattest/sharpest pitches are recorded during an earlier pre-processing step, and the number of transpositions in both the sharp and flat direction can be calculated so that, at most, there are two flats or two sharps in the piece. This avoids overly complex and non-realistic augmentations.

By transposing the dataset in such a practical way, the number of scores available to use during training effectively quadruples. Figure 21 visualizes the distribution of the number of times a piece can be transposed for the entire dataset. The bars are the sum of the flat-wise and sharp-wise transpositions. On average a piece can be transposed 11 times, with a small portion of pieces falling below 5; these pieces are highly **chromatic**, modulating to many keys.

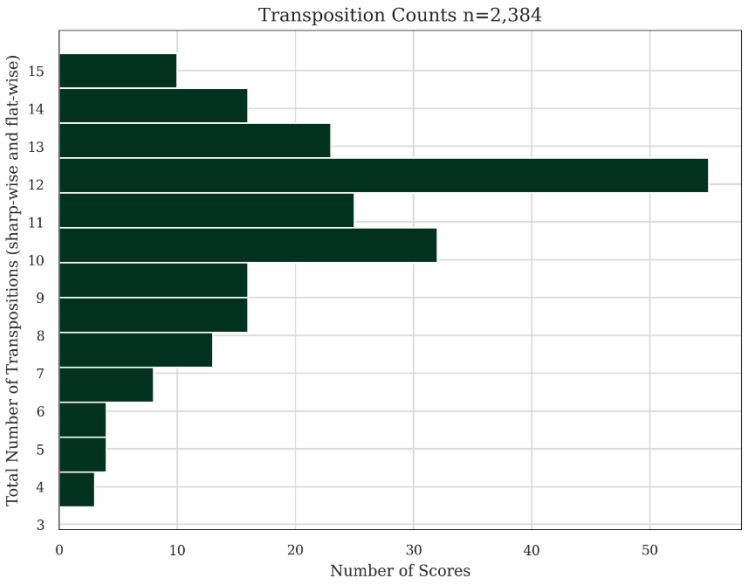

Figure 21. Distribution of transposition counts

3.5 Network Architecture

Most of this project's model architecture largely follows the Encoder portion of Vaswani et al's Transformer, where most of the learnable weights are within the Encoder stacks. Figure 22 contains a diagram of the project's model architecture; the input pitch matrix with the bass notes appended is passed through a max pooling layer with a pool size of 4, which decreases the frame time-step frequency to eighth notes from 32nd notes.

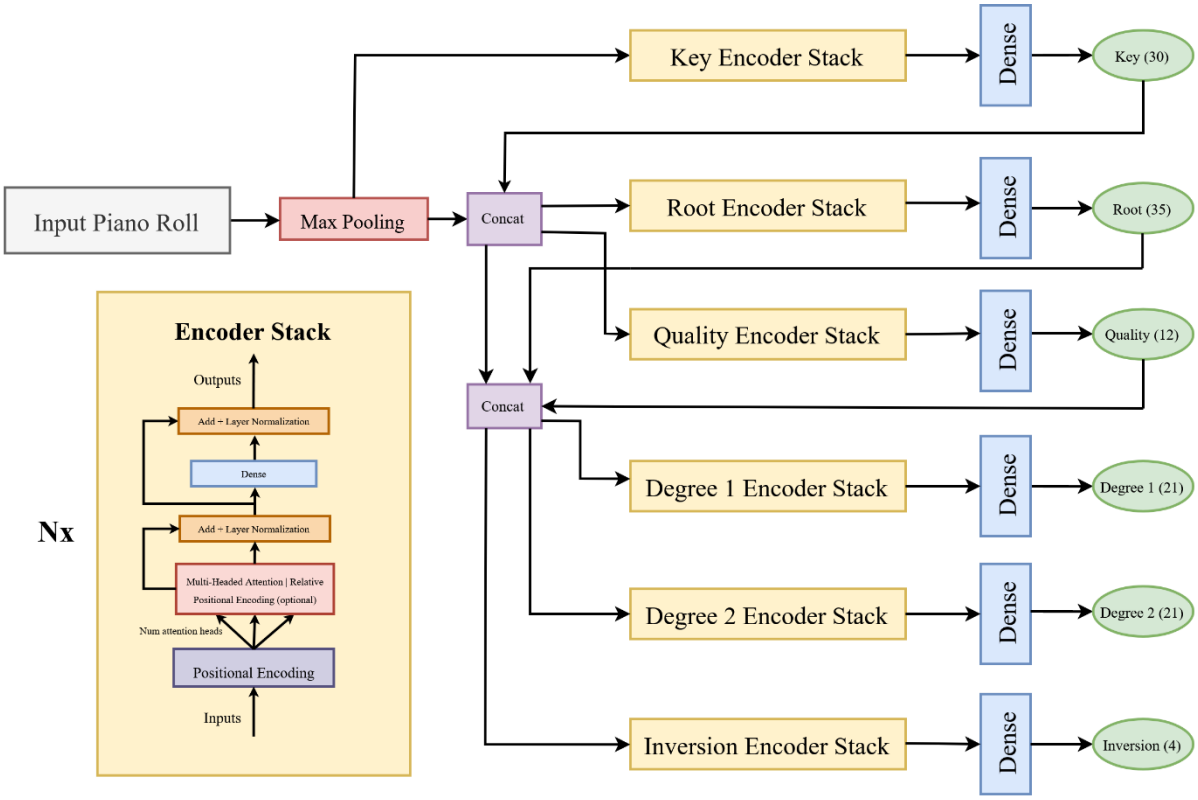

Figure 22. Multi-task encoder architecture with optional relative positional encoding for Roman Numeral classification.

Each task has its own encoder stack with a varying number of both encoder layers and self-attention heads, the numbers of which were determined via experimentation. Within the encoder stack's attention mechanism, either relative or absolute positional encoding can be used, depending on which suits each task's performance.

The outputs of certain tasks are concatenated, or chained, to the piano roll input to provide contextual information and additional features to subsequent tasks that may benefit from knowing previous predictions for congruency's sake. The nuances of the architecture and its chained connections will be further explored in the results section.

3.6 Model Implementation and Evaluation

The model's hyperparameter space (HP) is particularly large, because each encoder stack can be configured with positional encoding type, number of attention heads, number of encoder layers, the size of the internal dense connections, dropout rates, etc.. With an expansive HP optimization space and a very bespoke codebase not compatible with common HP optimization frameworks, a custom grid search was performed randomly across over 40 different options.

3.6.1 Experimental Settings

The models were trained using the Adam optimizer (Kingma & Ba, 2014) with a varying learning rate. Both an exponential decay approach and a custom rate scheduler were considered.

Figure 23 plots examples of both methods; the custom scheduler increases the learning rate linearly for the first *warmup_steps* and decreases it thereafter proportionally to the inverse square root of the step number. (Vaswani et al., 2017)

The other line corresponds to the exponential decay method, where an initial learning rate of 0.001 was decreased exponentially at a rate of 0.90. The equation to calculate the learning rate for each step can be found in Equation 1.

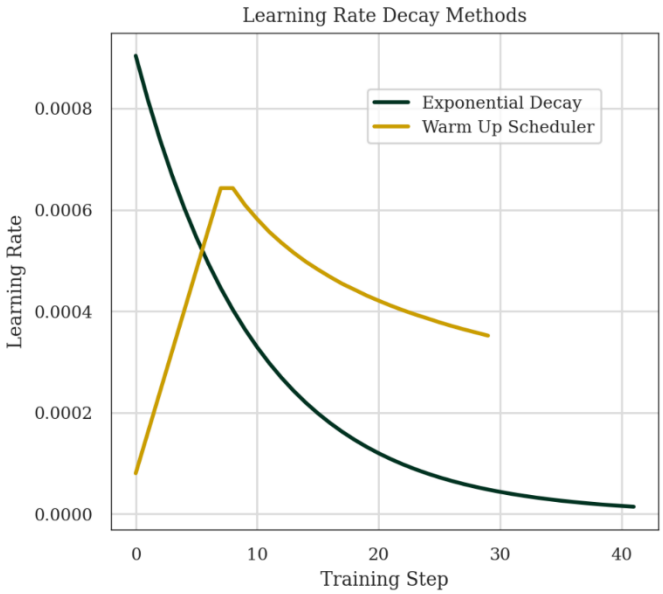

Figure 23. Examples of learning rate decay methods from experimentation process

$$lrate_{step} = lrate_{initial} * rate_{decay}^{\frac{step}{decay\ steps}}$$

Equation 1. Formula for calculating exponentially decreasing learning rate

The custom scheduler followed the same general formula as was introduced in Vaswani et al's paper to train the Transformer. This method is advantageous for models that can easily diverge in the early stages of training with large learning rates; the warmup period allows the adaptive learning rate optimisers to tune attention mechanisms and converge to global minima in a much more stable way. This stability is achieved by regulating the size of the parameter updates in the preliminary training steps. (Ma & Yarats, 2021)

3.6.2 Evaluation Metrics

The models are evaluated primarily by their Roman Numeral + Inversion accuracy, which requires all components of the following formula are correct:

$$\boldsymbol{RN} = \boldsymbol{Degree}\,\mathbf{1}\,\&\,\mathbf{2} + \boldsymbol{Key} + \boldsymbol{Quality} + \boldsymbol{Inversion}$$

Equation 2. Roman numeral annotation formula

Where the accuracy is calculated as the RN true positives divided by the total number of predicted labels. The above is a model-wide accuracy measure. In addition to RN accuracy, each individual task's accuracy will be analysed on its own, essentially as its own model. Precision, Recall, and F1 Score metrics will be utilised.

3.6.3 Accuracy Variations

It is useful to group certain tasks together, like degree, for example. A formula for both degrees would be as follows:

$$Degree\ Accuracy = \frac{\sum \boldsymbol{DG1\_TP} * \boldsymbol{DG2\_TP}}{Num\ Predictions}$$

Where ***DG1_TP*** and ***DG2_TP*** are binary vectors indicating that *Degree 1* and *Degree 2* were correctly predicted. When multiplied together and summed, the numerator represents the predictions where both tasks are correct. A similar operation is performed for RNs with inversions, RNs without inversions, and secondary dominant chords.

The model itself predicts the root of the chord as its own task, but it is also possible to derive the root based on the predicted key and the degree. Micchi et al. thought it would be of interest to look how often the derived label agrees with the independently predicted root; this concept is referred to as root coherence and will be evaluated along with task accuracies.

	actual positive	actual negative
predicted positive	TP	FP
predicted negative	FN	TN

Figure 24. Example of a confusion matrix from a binary classification task as conveyed by (Davis & Goadrich, 2006)

3.6.1 Precision, Recall, F1 Score

In addition to accuracy metrics, other performance metrics like precision, recall, and F1 will be used. In a decision problem classifier, labels are either positive or negative. The decision made by the classifier can be represented in a structure known as a confusion matrix or contingency table. True positives (TP) are predictions correctly labelled, false positives (FP) are negative

$$\text{Recall} = \frac{TP}{TP+FN}$$

$$\text{Precision} = \frac{TP}{}$$

$$F1 = \frac{2tp}{2tp + fp + fn}.$$

Figure 25. Formula for F1 score as displayed by (Lipton et al., 2014)

examples misclassified as positive, true negatives (TN) are those correctly labelled as negative, and false negatives (FN) refer to positive records predicted incorrectly as negative. (Davis & Goadrich, 2006)

Precision is calculated as the ratio of TP to TP + FP and represents the fraction of true positive records to all predicted positive records. This measure conveys the model's ability to distinguish only the most relevant data points. Recall is calculated by dividing the TP by TP + FN and is a measure of the model's ability to correctly identify all the relevant instances within the dataset.

The F1 score is a metric that uses both of these metrics within it; it is defined as the harmonic mean of precision and recall, combining the two metrics in to one. This project will use a weighted F1 score as each of the task's outputs are multi-class as opposed to binary. The weighted average of precision and recall is calculated across each task's outputs independently, and then averaged according to each label's prominence in the dataset. Figure 25 shows one definition of the metric only using true and false positives, but it can also be conveyed as $2 * \frac{(\text{Recall} * \text{Precision})}{(Recall + Precision)}$. This metric is useful for measuring the success of a classifier when one or several classes are rare in the dataset. F1 in addition to the other two constituent metrics will be utilised to evaluate this project's final model.

4. Results

4.1 Overview

This section pertains to the outcomes of the methods and experimentation processes laid out in **Section 3.0 Methods** to find the optimal model architectures and ultimately answer the research question: how components like self-attention and relative positional encoding can be applied to RN automatic harmonic analysis and their respective efficacies. It begins by summarising the model development and architectural experimentation processes; it then provides motivation for concatenating the outputs of certain tasks to the input matrices for other tasks in order to increase prediction coherence.

An analysis of the model's performance to various architectural hyperparameters is conducted, and the inter-task relationships are further expounded upon. The model's overall performance is reported and compared with the performance of Micchi et al's best performing model for the same dataset. An error analysis is carried out, where insights are gleaned regarding the common types of misclassifications for different tasks and how they can be interpreted harmonically.

4.2 Implementing Encoders with Self-Attention

An extensive code audit of Micchi et al's functional-harmony repository was conducted prior to any development work; this was to ensure feasibility of the project's outcomes and inform the researcher on the dataset's properties when pursuing possible modelling options.

The model development process began with a rough implementation of Google's *Transformer Model for Language Understanding*, a tutorial from TensorFlow, using Micchi et al.'s pre-processing framework. The Transformer has an Encoder / Decoder structure, and because RN analysis is a classification problem, not a translation problem, the Decoder portion is not applicable. Consequently, a search for similar implementations online was conducted to inform best practices in network architecture development. Transformer-derived implementations were found using spectrogram matrices as inputs for the task of music genre classification, but not for this project's task of functional harmony classification.

4.2.1 Shared vs Task-Specific Encoder Stacks

Early variations of the model architecture involved one shared encoder stack between each of the tasks. Figure 27 reiterates the final model architecture; initially, only one yellow stack was present. It only later occurred to the author that attention weights should be learned for each individual encoder stack. Consulted implementations of an encoder-like architecture related to music classification were not framed in a multi-class manner. This development proved to be advantageous, because individual task accuracy differed greatly with varying number of attention heads, and the number of encoder layers.

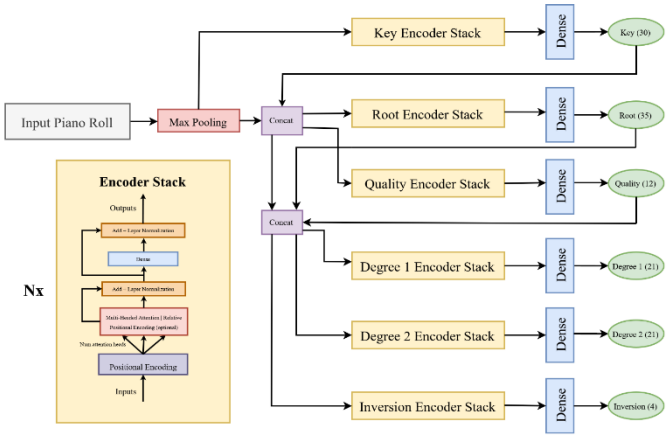

Figure 27. The multi-encoder stack architecture. Initially only one encoder-stack was used and shared among the six tasks.

4.3 Hyper-parameter search

To optimize the model's performance, a random grid search was conducted across 38 different hyperparameters, each containing an array of options. This search included parameters related to the model's architecture, e.g., the number of encoder layers or self-attention heads, as well as optimization options like the learning rate, its scheduling method, and mini-batch size.

One of the ancillary benefits of running thousands of training runs was the subsequent ability to learn how the different tasks correlated with each other. For instance, when the *Key* encoder model's performance is relatively high, so is the *Quality*'s accuracy. This was confirmed in the subsequent grid search analysis; the two tasks share an accuracy correlation coefficient of 0.82. This is presumably because they are looking at the same patterns and benefit from the same weight updates.

Figure 28 contains an example of a model's training run; each of the 6 task's training and validation accuracies are plotted. A limitation of the dataset can be observed in the Degree 1 accuracy time series. Because secondary dominant chords are used very infrequently in favour of those belonging to the tonic key, the distribution of non-diatonic chords vs diatonic ones is very skewed in favour of the latter. If the Y axis scale of the Degree 1 chart is examined, one can see that the accuracy is above 90% while others are near 70. This is the diatonic chord imbalance making itself known. Luckily, the model is still

able to incrementally improve and learn to improve its predictions regardless; but a stratified sample via augmentation in future research could alleviate this imbalance.

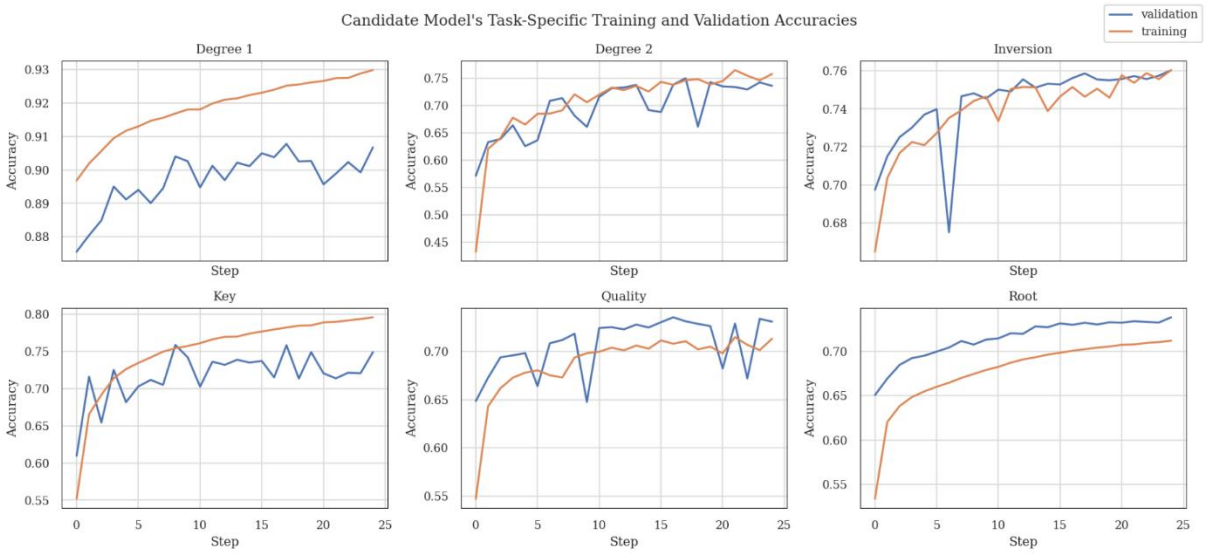

Figure 28. Each task's validation and training accuracies for a high-performing model candidate. This candidate's overall RN+Inversion accuracy was 39.47%.

4.3.1 Encoder Layers

Determining the optimal number of encoder layers in each task's encoder stack played a large part in the model optimization process. Certain tasks responded better to more encoder layers than others. For reference, Vaswani et al. included six encoder blocks in their Transformer model, which they do not provide a specific motivation for outside of experimentation. Across options ranging from 2 to 8 in increments of 2, they, too, saw diminishing returns after reaching a certain threshold. (Vaswani et al., 2017)

Key		
Num. Encoder Layers	**Avg. Accuracy**	**Num. Models**
1	75.85%	43
4	77.87%	30
6	77.30%	49
7	77.21%	11
10	43.17%	12

Table 2. Average accuracy for the Key task across varying number of encoder layers

Table 2 contains a summary of validation accuracies across various numbers of encoder blocks for the *Key* task. Accuracy saw diminishing returns after increasing the number of self-attention layers more than seven. Each task had their own optimal number of encoder layers. Both *Key* and *Root* benefitted from larger N when configuring the depth of their encoder stacks, with depths of 5 and 3 for each task, respectively, returning optimal classification accuracy. This contrasts with other tasks like *Quality* and *Degree* that performed best with only one encoder layer. This is likely because *Key* and *Root* require more of the input sequence to correctly classify when compared to other tasks, and subsequently benefit from more network connections that can learn more nuanced relationships.

4.3.2 Attention Heads

Even more so than the number of encoding blocks, the number of attention heads varied greatly between each of the tasks' encoder stacks. To refresh, models like the Transformer and BERT use multiple attention mechanisms in parallel, with each attention head potentially focusing on different parts of the input, which in turn makes it possible to express sophisticated functions beyond the simple weighted average. (Michel et al., 2019)

Similar results to those of Michel et al were obtained during the model hyperparameter tuning process regarding the number of attention heads and its impact on task performance. For example, the accuracies of certain tasks like *Degree 1* and *Degree 2* were found to be negatively correlated with the number of attention heads in their attention mechanisms. With a correlation coefficient of -0.201, the overall Degree accuracy (where both degrees must be correctly predicted) did not benefit from larger amounts of attention heads. This contrasts with the *Quality* task, with an accuracy correlation coefficient of 0.247; indicating that *Quality* accuracy tends to increase when the number of attention heads in its encoder stacks also increases.

Key		
Num. Attention Heads	**Avg. Accuracy**	**Num. Models**
2	68.13%	35
5	73.92%	147
7	71.26%	125

Table 3. Candidate model Key accuracies from grid search over three possible attention head values

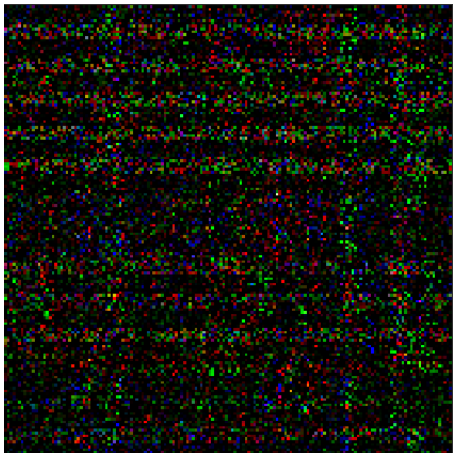

Figure 29. An example of attention weights obtained during the learning process for the Key task. The grid-like structure of attention-scores could be indicative of rhythmic patterns and relevant strata of the pitch space.

4.3.3 Positional Encoding

As mentioned previously, the Transformer architecture itself is not inherently capable of making use of the sequential order of the input sequences. To remedy this, Vaswani et al introduced the concept of positional encoding in the Transformer paper, where information from a deterministic function is injected regarding the absolute position of the tokens for the entirety of the sequence. This enables the model to learn the sinusoidal pattern and consequently learn to identify the relevancy of positions when making predictions.

We hypothesised that absolute positional encoding would underperform relative positional encoding, due to its ability to focus on the time-space near the chord of interest. This hypothesis was rejected, in almost all situations save for a few odd training-runs, tasks using relative positional encoding underperformed relative to absolute. Table 4 contains a summary of a subset of the grid search training runs where only relative positional encoding was used. When using relative positional encoding, a hyperparameter referred to as "maximum distance" is required; this specifies the distance both forward and backward from the current chord that is not clipped. Across varying max distance options, when

		Avg. Accuracies				
Max. Dist	**Num. Models**	**Degree**	**Inversion**	**Quality**	**Key**	**Roman + Inv.**
2	15	60.69%	70.69%	67.83%	72.89%	32.24%
4	21	60.89%	69.54%	67.83%	71.43%	31.42%
6	15	60.68%	68.75%	67.99%	73.68%	32.10%
8	13	60.95%	69.91%	67.12%	75.65%	33.08%
10	14	60.72%	70.13%	67.81%	74.99%	32.95%
16	17	61.24%	70.12%	68.07%	76.07%	33.46%
32	19	62.05%	70.60%	69.78%	77.30%	34.97%
64	11	62.55%	71.34%	70.40%	77.97%	35.93%
128	15	63.10%	71.52%	70.57%	78.29%	36.42%
Total	**140**	**61.40%**	**70.24%**	**68.57%**	**75.18%**	**33.51%**

Table 4. Average task and RN accuracies for models utilising only relative positional encoding across a range of max-distance options.

the attention heads were able to see more time steps in either direction, overall RN accuracy increased in lockstep. Most tasks saw marginal improvements with larger distances, but the *Key* task is the best example; this result is intuitive because the tonic of a piece of music cannot be determined from one chord on its own.

A conclusion can be made that all RN tasks make better use from being able to analyse the entirety of the input sequence, and that local chord information does not provide any information that can't be gleaned elsewhere. The final model architecture did not make use of relative positional encoding for any task, and instead utilised absolute encoding as used in the *Attention is All You Need* paper. Various maximum encoding lengths were tested, but an encoding length 4x the original time dimension of the input sequence proved to be the most effective.

4.4 Final model selection

The grid search and model refinement process took place over several weeks; using TensorBoard as a tool to evaluate individual task accuracies in real-time, the hyperparameter space was narrowed throughout the process. Table 5 lists the 36 final hyperparameters and their corresponding values determined via randomized grid search. The number of encoder layers determines the model's size. The best obtained model contains total of 16 encoder layers across all tasks, each containing self-attention heads within them. The final model contains 2,724,565 learnable parameters, 30x more than Micchi et al.'s best model.

Best Model Hyperparameters	
Hyperparameter	***Value***
Batch Size	32
Chunk Size	160
Num. Features	70
Training Epochs	250
Initial Learning Rate	0.001
Beta 1	0.95
Beta 2	0.95
Epsilon	1.00E-09
Learning Rate Scheduler	Exponential
Label Smoothing Value	0.5275
Dropout Rate	0.33
Encoder Dense Units	512
Post-Encoder Dense Units	128
Max. Pos. Encoding	640
Max. Distance	N/A
Pool Type	Avg
Key Chain	True
Quality Chain	True
DG1 Encoder Layers	2
DG2 Encoder Layers	2
Quality Encoder Layers	2
Inversion Encoder Layers	2
Key Encoder Layers	5
Root Encoder Layers	3
DG1 Attn. Heads	3
DG2 Attn. Heads	3
Quality Attn. Heads	4
Inversion Attn. Heads	1
Key Attn. Heads	1
Root Attn. Heads	1
DG1 Rel. Pos. Encoding	False
DG2 Rel. Pos. Encoding	False
Quality Rel. Pos. Encoding	False
Inversion Rel. Pos. Encoding	False
Key Rel. Pos. Encoding	False
Root Rel. Pos Encoding	False

Table 5. Final hyperparameters and their corresponding values for the best performing model achieved during grid search

In addition to the learning rate parameters like epsilon, beta, etc., various early-stopping criteria were explored with the intention of promoting task accuracy agreement. Initially, validation binary cross entropy loss was used as the early-stopping criteria, but this proved to fail when 4/5 tasks would perform well, and another would diverge (even if temporarily), and the entire model would stop training due to lack of loss not decreasing after a certain number of epochs.

Validation Accuracy Correlation Matrix

	Key	Degree 1	Degree 2	Quality	Inversion	Root	Degree	Secondary	Root Coherence	Roman	Roman + Inversion
Key	1.00	0.59	0.58	0.61	0.52	0.72	0.58	0.53	0.86	0.84	0.83
Degree 1	0.59	1.00	0.85	0.79	0.64	0.70	0.87	0.88	0.78	0.81	0.82
Degree 2	0.58	0.85	1.00	0.82	0.60	0.70	0.99	0.90	0.86	0.87	0.88
Quality	0.61	0.79	0.82	1.00	0.75	0.77	0.82	0.72	0.75	0.81	0.82
Inversion	0.52	0.64	0.60	0.75	1.00	0.65	0.58	0.52	0.54	0.59	0.62
Root	0.72	0.70	0.70	0.77	0.65	1.00	0.70	0.64	0.72	0.72	0.72
Degree	0.58	0.87	0.99	0.82	0.58	0.70	1.00	0.91	0.87	0.89	0.90
Secondary	0.53	0.88	0.90	0.72	0.52	0.64	0.91	1.00	0.80	0.81	0.81
Root Coherence	0.86	0.78	0.86	0.75	0.54	0.72	0.87	0.80	1.00	0.97	0.97
Roman	0.84	0.81	0.87	0.81	0.59	0.72	0.89	0.81	0.97	1.00	1.00
Roman + Inversion	0.83	0.82	0.88	0.82	0.62	0.72	0.90	0.81	0.97	1.00	1.00

Figure 30. The Pearson correlation coefficients of various accuracy metrics used during model evaluation.

With the aim of finding a new stopping criterion, an analysis was undertaken to discern which of the task accuracies performed best when RN accuracy was at its highest. Figure 30 displays the inter-task accuracy relationships in addition to model-wide metrics like RN accuracy and root coherence. With a correlation coefficient of 0.88, the Degree 2 task appeared to contribute the most to the overall model's performance and was subsequently used as stopping criterion as the code could not evaluate full RN accuracy during the training process, as it is calculated after the training process is completed. This proved to be a fruitful decision, as model performance improved significantly once Degree 2 Accuracy was chosen as the stopping criterion with a patience of seven epochs and a minimum delta of 0.0001.

4.4.1 Task Relationships

The idea to concatenate the output predictions of other tasks to the inputs of others was originated when analysing how certain tasks learned similarly to one another. Not only were the hyperparameters tuned during the grid search process, but so was the architecture. It was observed that certain tasks tend to perform well together, like *Quality* and *Degree 2*. This is presumably because the quality of a chord is implicit in its scale degree, like how the IV chord of a major key always has a major third if played diatonically. Similarly, *Key* and *Root* learned similarly to one another, which aligns with the author's preconceptions.

The model optimisation process proved to be difficult prior to chaining tasks to other tasks; no matter how the parameters were tweaked, when certain tasks would benefit after making a parameter

adjustment, others would decrease, and overall RN accuracy would suffer. In essence, certain tasks' performances were mutually exclusive with others.

In Figure 30, it can be observed that *Key* and *Inversion* do not perform that well independent of one another, as they have the lowest inter-task accuracy correlation coefficient of 0.52. Similarly, *Degree 2* (which was the most correlated with overall RN accuracy) also had a low *Key* correlation coefficient: 0.58. This task divergence led to the idea that the outputs of the *Key* task should inform every other task, because this holds true in music theory. If a musician knows what key a song is in, they are then able to play a chord built upon a certain scale degree. Chaining the outputs like this was the first instance of explicitly building a concept from music theory into the model itself.

Selected Hyperparameter Value and Accuracy Correlation Matrix

	Key	Degree 1	Degree 2	Quality	Inversion	Root	Degree	Secondary	Roman	Roman + Inversion
Quality Attn. Heads	0.22	0.25	0.27	0.25	0.20	0.25	0.28	0.27	0.29	0.30
Inversion Encoder Layers	0.13	0.21	0.27	0.23	0.17	0.24	0.27	0.20	0.21	0.20
Post-Encoder Dense Units	0.14	0.06	0.15	0.10	0.00	0.02	0.15	0.16	0.18	0.18
Batch Size	0.19	0.06	0.12	0.14	0.11	0.19	0.11	0.05	0.17	0.17
Inversion Attn. Heads	0.09	0.17	0.18	0.15	0.08	0.12	0.18	0.17	0.17	0.17
DG1 Encoder Layers	0.03	0.20	0.21	0.19	0.13	0.14	0.21	0.21	0.14	0.15
Quality Encoder Layers	-0.03	0.27	0.24	0.20	0.18	0.20	0.25	0.22	0.09	0.10
Dropout Rate	-0.02	0.10	0.03	0.02	0.05	-0.02	0.04	0.12	0.05	0.06
DG2 Rel. Pos. Encoding	0.02	-0.02	-0.03	-0.06	-0.03	-0.03	-0.04	-0.03	0.00	-0.01
Key Rel. Pos. Encoding	-0.03	-0.03	-0.02	-0.01	-0.06	0.04	-0.02	0.06	-0.01	-0.02
DG1 Rel. Pos. Encoding	-0.01	-0.02	-0.01	0.02	0.04	0.02	-0.02	-0.03	-0.03	-0.03
Key Attn. Heads	0.05	-0.17	-0.13	-0.11	-0.11	-0.08	-0.15	-0.12	-0.03	-0.04
Chunk Size	-0.03	-0.05	-0.04	-0.06	-0.06	-0.04	-0.03	-0.08	-0.08	-0.08
Quality Rel. Pos. Encoding	-0.08	0.06	-0.02	-0.12	-0.11	-0.07	-0.02	0.04	-0.10	-0.11
Inversion Rel. Pos. Encoding	-0.15	-0.06	-0.11	-0.16	-0.18	-0.15	-0.10	-0.05	-0.15	-0.17
Encoder Dense Units	-0.15	-0.09	-0.12	-0.13	-0.07	-0.14	-0.13	-0.09	-0.17	-0.17
DG2 Attn. Heads	-0.16	-0.08	-0.21	-0.15	-0.12	-0.15	-0.20	-0.13	-0.21	-0.21
DG1 Attn. Heads	-0.14	-0.08	-0.21	-0.16	-0.12	-0.12	-0.20	-0.14	-0.21	-0.22
Key Encoder Layers	-0.24	-0.08	-0.13	-0.07	-0.05	-0.06	-0.12	-0.14	-0.24	-0.25
DG2 Encoder Layers	-0.22	-0.10	-0.18	-0.07	-0.06	-0.07	-0.18	-0.20	-0.25	-0.25

Figure 31. Pearson's correlation matrix of Individual task and model-wide accuracies with selected hyperparameter values

As seen in Figure 31 that contains more correlation coefficients, but with the values of selected model hyperparameters with task and model-wide accuracies, *Degree 1* tended to perform better when key predictions were included in its encoder's input vector. Consequently, the *Secondary* metric (the accuracy of chords borrowed from other keys) increased, too. As stated previously, the Degree 1 task suffers from class imbalance as chords from borrowed keys, or secondary dominant chords, are relatively rare. The *Key* and *Degree 1* are inherently related, because the *Key* of a chord changes when it borrows from another pitch centre, and as such, these two tasks need to agree with one another.

4.4.2 Model vs Task-Level Hyperparameters

Refining the hyper-parameters of each individual task proved to be difficult. Because of how the model was integrated in to Micchi et al.'s pre-processing pipeline, certain training hyper-parameters could only be tweaked at the model level, as opposed to the task level. For instance, the label smoothing value,

used to reduce the model's over-confidence in multi-task classification, affected each of the tasks differently. The *Key* task is unique from the others, because it received a greater variety of classification label options directly from the data augmentation process via transposition, whereas the other tasks did not, because they rely on a more local context of a piece of music. Because each of the model's tasks exhibited its own unique behaviour in relation to model-level hyperparameters, a balanced had to be struck for each, where the overall model performance is used as the determining factor when deciding on a final hyperparameter value. With more time to refactor the codebase and use newer versions of Python libraries, a newer model training framework could be developed that allowed for optimization of more parameters that were limited to only the entire model.

4.5 Model Performance on validation data

	Key	Degree	Quality	Inversion	RN
Multi-Task Encoder Net	80.6	66.1	73.9	**73.9**	41.3
Micchi et al (2020)	**82.9**	**68.3**	**76.6**	72.0	**42.8**
Difference	-2.3	-2.2	-2.7	**1.9**	-1.5

Figure 32. A comparison of the percent accuracy between this project's best performing model and the best model from Micchi et al.'s Not All Roads Lead to Rome paper.

This project used the same training and validation data as was utilised in the baseline paper: 0.9 and 0.1, respectively. Figure 32 reports on the RN and individual task validation accuracies of this project's best-performing model in comparison to those reported by Micchi et al. (Micchi et al., 2020) The *Degree* task is only correct when the predicted labels match the validation record for both Degrees 1 and 2. RN is correct when each of the previous four columns are correctly predicted.

	Key	Degree 1	Degree 2	Quality	Inversion
Precision	85.04	88.37	72.09	72.75	72.21
Recall	80.76	90.03	72.95	74.19	73.87
F1	82.04	88.32	72.06	72.71	71.51

Figure 33. Weighted average Precision, Recall, and F1 Score for each individual task's output. These metrics were only calculated for the outputs of each task, in comparison to Figure 30 which contains derived accuracy measures like RN and Degree

Figure 33 reports each task's weighted average precision, recall, and F1-Scores which are defined in Methods section of this report. Of the five tasks listed, the *Key* task's precision and recall scores are of interest, due to its precision score being 4.28% higher than its recall. When the model predicts a chord's *Key*, for each of the possible classes, the ratio of TP to TP + FP (precision) is higher than the ratio of TP to TP + FN (recall). When the model predicts a certain class, 85.04% of the positive predictions were correctly classified and 80.76% of what the model should have predicted correctly, were. This indicates that the model is relatively adept at predicting positive classes for the *Key* task but does poorly

when compared to what it should have predicted. Because the *Key* task benefitted the most from data augmentation via transposition, this result is surprising, as the likely culprit of this difference is class imbalance.

Developing a model that consistently predicted the 4 tasks well proved difficult. Certain architecture choices improved certain tasks, while decreasing others, and subsequently had a negative impact on the overall RN accuracy. An example of this was observed with the dimension of the final hidden feed forward neural network layer; Figure 34 reports on the average model and task accuracies when varying this architectural parameter between 64 and 128 neurons. When using a smaller number of neurons, the model was able to predict the *Inversion* of a chord 3% better but predicted its *Degrees* and *Quality* 6% and 5% worse, respectively. Further development is required to fully optimise the training process to favour models that can learn to predict each of the tasks in a congruent manner.

Final Dense Dimension	Key	Degree	Quality	Inversion	RN
64	68.48	50.80	60.67	**68.55**	25.25
128	**69.68**	**54.21**	**63.75**	66.67	**27.84**

Figure 34. A comparison of task and model accuracies between two dimension options for the final dense feed forward layer prior to the softmax activation layer of the network. It should be noted these data are from hundreds of training model runs that resulted in low averages.

Unfortunately, this project's architecture experimentation process failed to yield an overall RN accuracy higher than Micchi et al.'s results achieved in 2020 but improved upon inversion accuracy. One of the advantages of their design compared to the encoder-based model is the ability to preserve the same feature maps learned via convolution for multiple convolutional layers to analyse the same information at successive levels of abstraction. (Micchi et al., 2020) The closest component of this project's model architecture would be the encoder stacks themselves. Micchi et al.'s usage of DenseNet (Huang et al., 2016) in their model likely provided more general learning more suitable to learning rhythmic patterns over longer distances than the self-attention mechanisms could.

4.6 Common Annotation Mistakes

As mentioned in the introduction to this report, RN annotations and functional harmony interpretations are somewhat subjective, and can have multiple correct annotations. It was expected that a trained model would make classification mistakes like those that a human might make. This hypothesis was validated with several of the tasks after analysing common errors for each.

Figure 35 shows an illustrative example of common errors when predicting the key an individual chord belongs to for one of J.S. Bach's Preludes. The X axis is the time dimension represented in chord-frames, or time-steps. The Y axis contains the possible key predictions ordered by the number of accidentals in each key. Each of the red points corresponds to a false positive; every FP will have a corresponding false negative indicating what the correct label should have been. From the referenced figure, one can see that the piece modulates from d minor to a minor, and during that modulation period

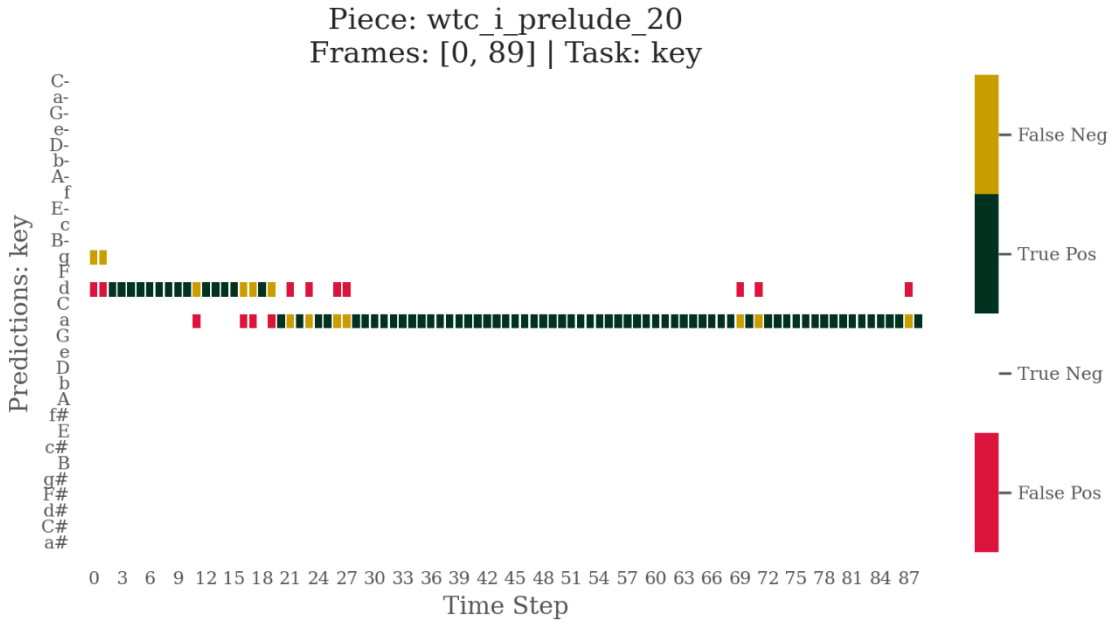

Figure 35. Time step error analysis for Key predictions of J.S. Bach's Prelude No. 20 from the Well-Tempered Clavier. Code used to generate this figure was adapted from Micchi et al.'s repository.

(around time step 20), the model fails to recognize the change in tonality for each chord, and only predicts some of them correctly. It makes logical sense that these keys would be mis-classified; this is because a minor and d minor are next to each other on the **circle of fifths**, indicating that they are very closely related and share the most amount of notes with each other than any other key. In the first few time-steps, it can be observed that g minor was predicted, which is on the other side of d minor in the circle of fifths; the two keys share all pitches except for E♭.

In Figure 36, a confusion matrix of the model's *Key* predictions vs their annotated labels is shown for the entire validation dataset. Here one can see the common misclassifications that the model makes. The X and Y axes of this chart are ordered in the circle of fifths for major keys, then for minor keys, to show how closely related the adjacent keys are to one another. For example, it is observed that the model frequently confuses A major for E major, which share all the same notes except for D# which is unique to E major.

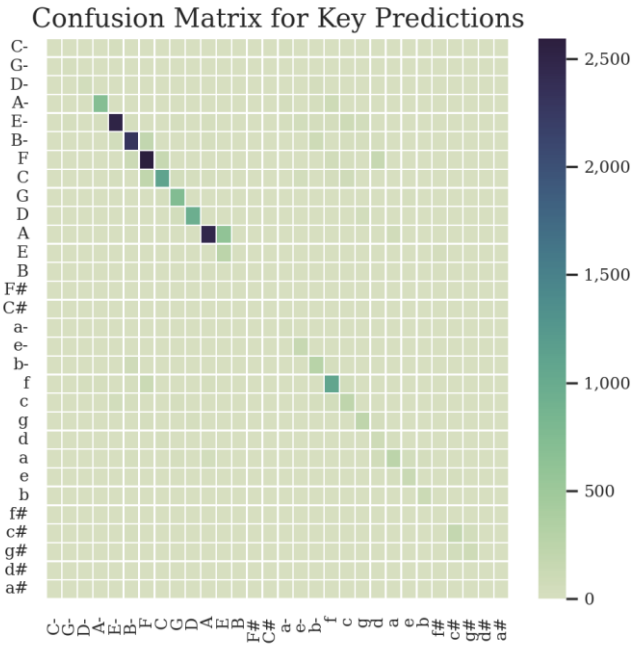

Figure 36. Classification confusion matrix for the Key task on all validation chord time-steps.

Figure 37 shows a similar error analysis for the *Quality* task of the same section of J.S. Bach's Prelude No. 20. An interesting trend can be gleaned from the figure; the most common misclassifications are between chords that share similar intervals. For example, during frames 20 to 25, diminished seventh chords are misclassified as minor triads. Diminished and minor chords both contain a minor third, with a diminished chord differing only by its fifth being one semitone lower. Because these chords are composed of similar intervals it is understandable why the system would struggle to distinguish them.

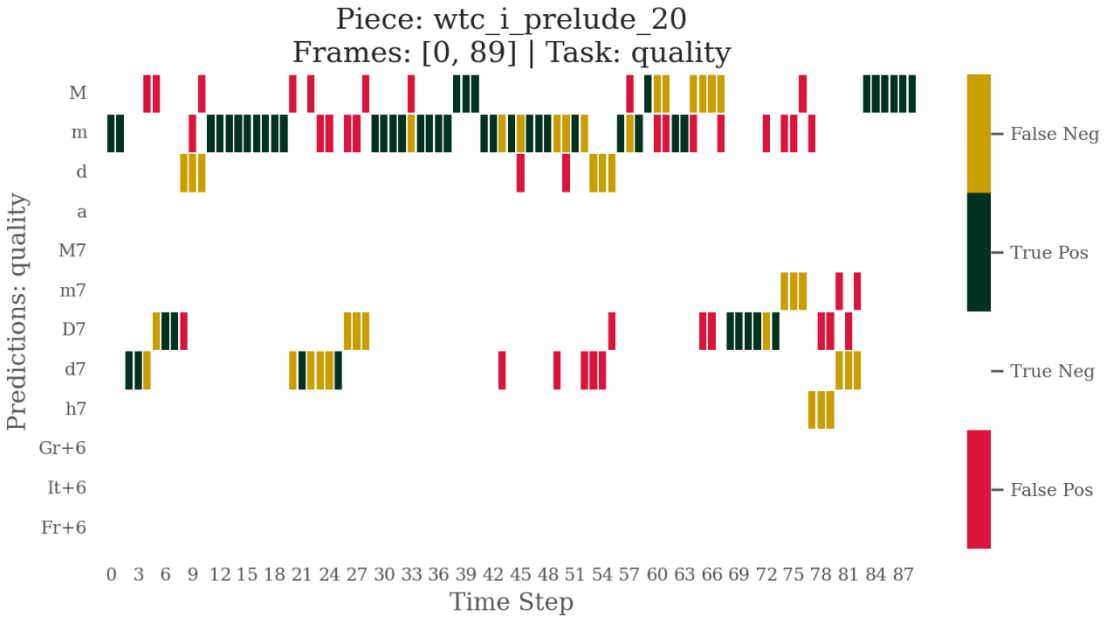

Figure 37. Time step error analysis for Quality predictions of J.S. Bach's Prelude No. 20 from the Well-Tempered Clavier. Code used to generate this figure was adapted from Micchi et al.'s repository.

As seen in Figure 38, a confusion matrix for the *Quality* task, major triads (M in the Figure) and dominant seventh chords (D7) are commonly mis-classified. These chords are identical except for the addition of a minor seventh interval in the case of D7 chords, which again aligns with expectations of common annotation disagreements in human analyses that would also be observed in a classification system. The relative sparsity of D7 chords to M chords also could contribute to the model overfitting to M.

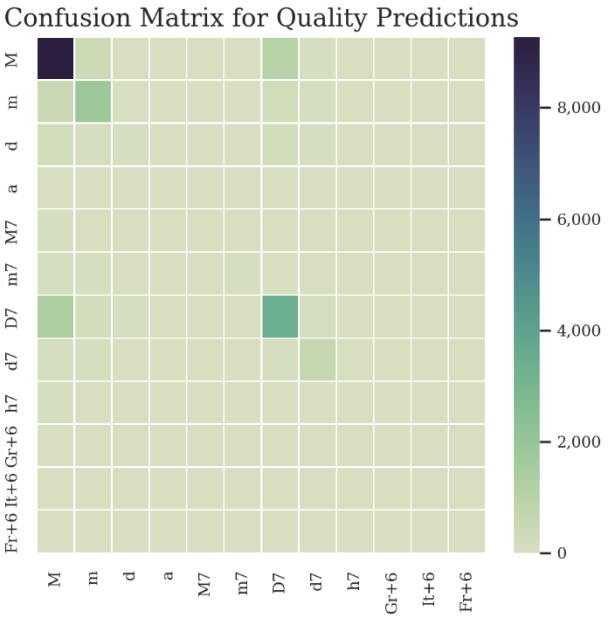

Figure 38. Classification confusion matrix for the Quality task on all validation chord time-steps.

Observed in the confusion matrix is the imbalance of chord qualities in the dataset; the dataset's constituent musical pieces primarily use major triads, much more than minor or diminished ones. This suggests further data augmentation would be beneficial.

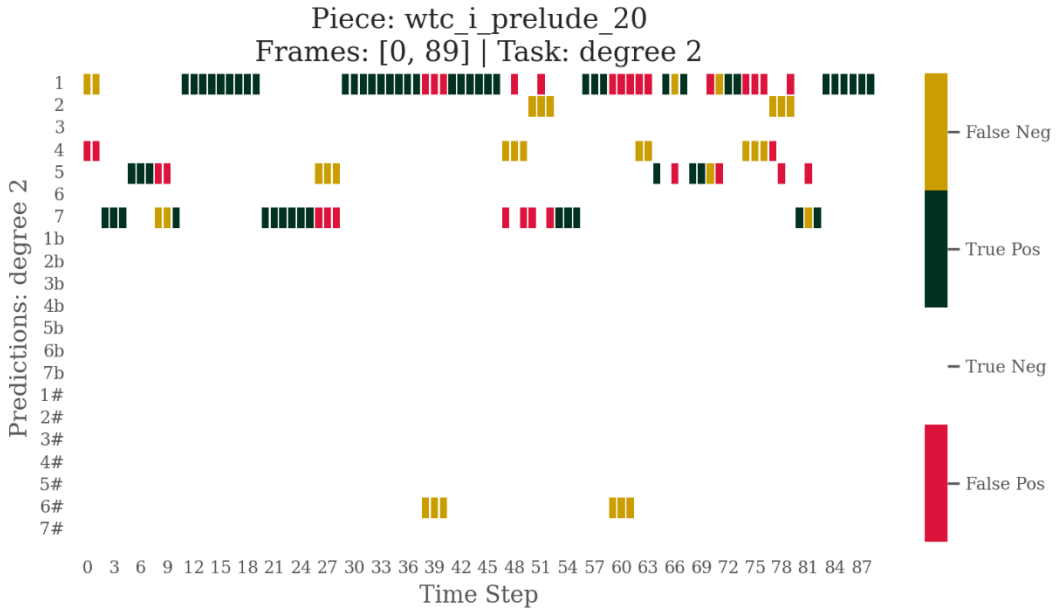

Figure 39. Time step error analysis for Degree 2 predictions of J.S. Bach's Prelude No. 20 from the Well-Tempered Clavier. Code used to generate this figure was adapted from Micchi et al.'s repository.

Figure 39 contains the same error analysis for the *Degree 2* task, the task that predicts the scale degree upon which the chord is built. The *Degree 2* misclassifications of the Bach prelude largely occur on the same time steps as the *Quality* misclassifications; this points to the system agreeing with itself, which is a desired trait, but the classifications are incorrect. In this example, the 7th scale degree is often confused with the 5th. In a minor diatonic context, the chords built upon these scale degrees are both

major, which, again, aligns with expectations in the types of chords that could be confused for one another.

As seen in the Degree 2 confusion matrix in Figure 40, for the entire validation dataset, the I and V chords are often confused, which again corroborates with the misclassifications seen from the *Quality* task—the M and D7 chords are commonly confused, and those chords are the I and V chords within the context of a major key. Additionally, one can observe the sparseness of non-diatonic chords. Any label beyond the first seven cells are chords that use tones outside of the key and are relatively rare.

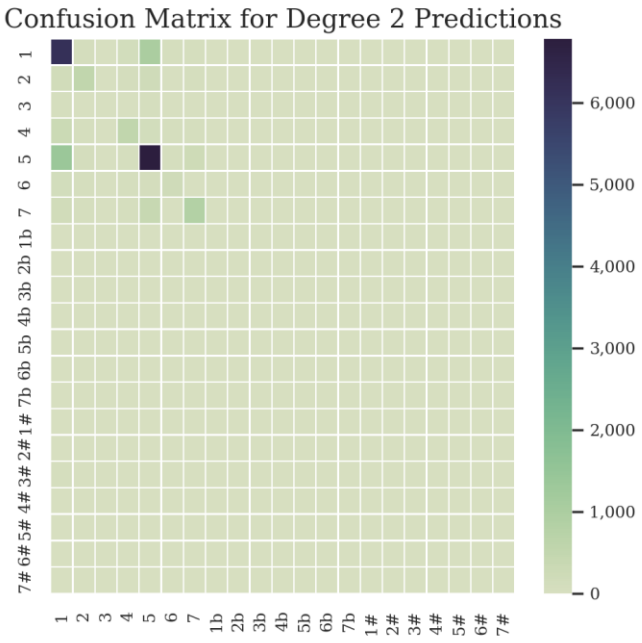

Figure 40. Classification confusion matrix for the Degree 2 task on all validation chord time-steps

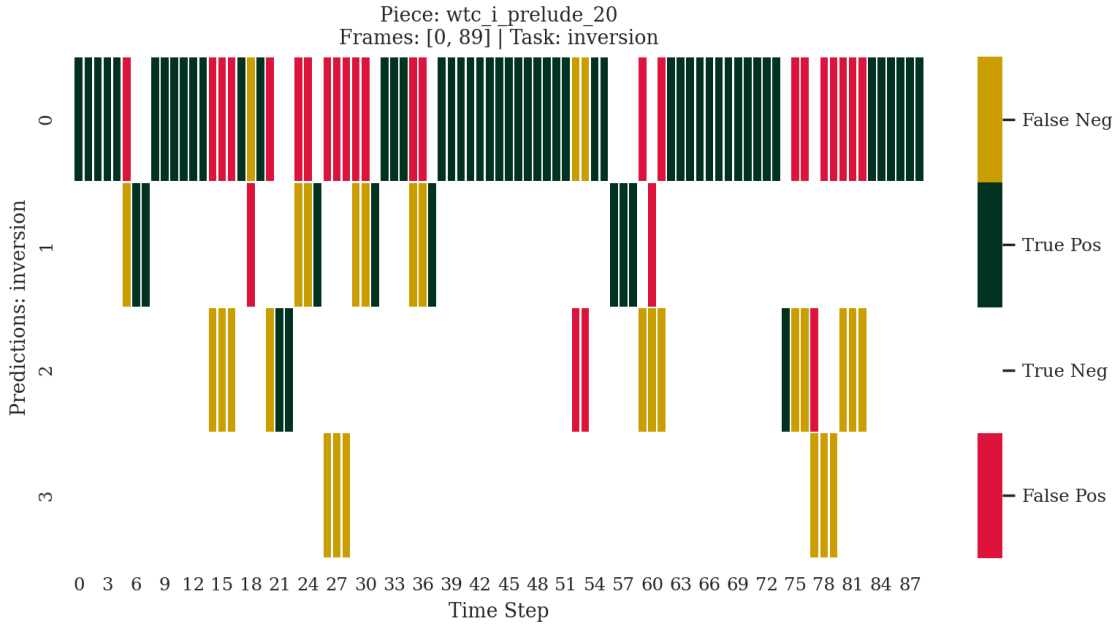

Figure 41. Time step error analysis for Inversion predictions of J.S. Bach's Prelude No. 20 from the Well-Tempered Clavier. Code used to generate this figure was adapted from Micchi et al.'s repository.

Figure 41 contains the error analysis for the same example input sequence. Here it is observed that this specific piece's chords are primarily in root position, i.e., the chord tones are ordered sequentially from lowest to highest in pitch. In this example, a misclassification trend is not discernible like the other analysed tasks. Figure 42 displays the misclassification errors at the validation dataset level for the *Inversion* task via its another confusion matrix. Again, most chords in the dataset are in root position as

evidenced in Section 3's Figure 18. This is evident in the top-left most cell of the matrix. Of the inversions, a 1st inversion triad, which has the third of the chord in the bass, is misclassified most frequently.

4.7 Conclusion

The results of this section demonstrate the efficacy of multi-headed self-attention in machine learning problems outside of natural language processing. Using a multi-task model with sequential prediction concatenation for certain tasks based on their musical relationships, comparable performance can be achieved with convolution-based approaches, though there is room for further improvement.

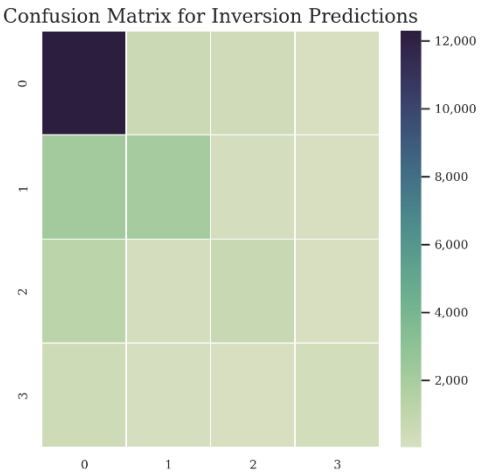

Figure 42. Classification confusion matrix for the Degree 2 task on all validation chord time-steps

5. Discussion

5.1 Evaluation of Objectives

5.1.1 Design, train, and deliver bespoke functional harmony model

A custom multi-task model architecture leveraging multi-headed self-attention was successfully designed and trained by leveraging existing pre-processing code developed for Micchi et al.'s *Not All Roads Lead to Rome* paper. This objective encapsulated the majority of this project's effort and required a steep learning curve in several areas.

Several Transformer-based models were recreated from various repositories and online tutorials to better understand how the complicated architecture is constructed, how it learns, and how it differs from convolution-based models for sequence classification tasks. Several months were spent attempting to deconstruct and adapt the code from Google's official Transformer repository; this process almost resulted in the abandonment of the project and would have required a change of topic. Thankfully, this obstacle was overcome by perseverance and excessive trial and error while integrating the RN encoder model to the *functional-harmony* repository's pre-processing pipeline.

Due to the random nature of the grid search and the high number of hyperparameter options at both the model and task-level, further model refinement could have been pursued; however, after obtaining an RN accuracy above 40%, the model's performance was more than high enough to confirm the validity of self-attention's usage outside of natural language tasks, and further performance optimization was deemed unnecessary for this project's purpose.

5.1.2 Performance analysis of final model and comparison to baseline

The Results section of this project serves as a performance analysis, as well as a comparison to the baseline convolution-based model; as such, this objective was successfully completed. Micchi et al.'s existing repository contained its own Python modules dedicated to the evaluation and comparison of model variations, which was very helpful in determining the optimal configuration.

By utilising TensorBoard's scalar and hyperparameter logging features, performance trends were easily extractable via correlation analyses of both hyperparameter values with overall metrics, as well as how each RN task performed relative to the other tasks. This inter-task analysis surfaced which tasks tended to perform well together, which tasks were inversely correlated, and subsequently resulted in further architectural development by concatenating the outputs of global scope tasks like *Key* to the inputs of local scope tasks like *Degree 2* and *Quality*. The concept of concatenating predictions from upstream tasks using both concepts from music theory and the performance trends to inform the architectural decisions is not something that was initially planned as part of this project; however, concatenating the

Key predictions to all upstream improved overall RN accuracy by 12%, which reiterates the importance of domain knowledge in conjunction with technical expertise when developing models for complex tasks.

The prediction confusion matrices illuminated the negative effects of class imbalance in the dataset. As seen in Figure 40, the prediction confusion matrix for the *Degree 2*, degrees 1 and 5 are by far the most predicted, because they are the most prevalent in the dataset, as is evidenced in Figure 19. As a consequence of this, the model doesn't learn to predict the other degrees as well as it learns to predict 1 and 5. Future iterations of this project would involve better handling for class imbalance, tailored for each task. Some class imbalance was mitigated by implementing and testing various label smoothing values to reduce the overall model's confidence in output predictions in the hopes that the model would learn to predict underrepresented tasks better. This exercise yielded modest improvements in performance but was only tailorable at the model-level.

Previous Roman numeral models do not use the F1 score as a performance metric, for reasons unknown to the author. During the early stages of this project, only task accuracies were analysed and monitored as that is what previous researchers have reported. However, a weighted average F1 score was better able to represent the model's ability at predicting all of the possible task labels, regardless of their distribution in the dataset and resulted in a much better indication of which tasks required the most improvement and further development.

As seen in Figure 32, which compares the accuracies of this project's best model against those of Micchi et al., three out of the four tasks underperformed in comparison, with *Inversion* being the exception. Inversion is the one task that in itself does not require sequential information to classify, it is merely the ordering of pitches in a chord. Micchi et al.'s best model architecture used many 1-D convolution layers, even for the tasks that require only local information like *Inversion.* This project's *Inversion* accuracy suggests that self-attention based sequential classifiers perform better than a convolutional GRU equivalent for tasks that do not have long-term sequential dependencies like *Degree,* which requires the contextual information from the current *Key*.

Because RN accuracy requires all four tasks to be correctly classified, a decrease in one task's performance has a large impact on overall performance. Of the three tasks that performed worse than Micchi et al.'s best model, on average, their accuracy deltas were -2.4%. With three out of four tasks underperforming, one would naively think that overall RN accuracy would be significantly lower, because the tasks aren't necessarily related, and this might have been the case if outputs from certain tasks weren't concatenated to upstream ones. Achieving an overall RN accuracy 1.5% lower than the baseline while three of the four tasks performed even worse signals that the three tasks predict consistently with each other, but on an individual task basis, they underperform. This is evidence of

upstream task output concatenation improving the model's ability to supply predictions that agree between tasks.

5.1.3 Answering the research question

When evaluating the results against this project's research question, "**To what extent can components from neural network architectures such as the Transformer and other recently introduced machine learning methods be applied to automatic Roman numeral automatic harmonic analysis?**", several insights can be gleaned.

The results of this project validate that models with self-attention mechanisms are capable of learning complex sequential relationships in contexts outside of natural language tasks. They surface that an encoder-based model is better able to predict a chord's inversion but fails to outperform the other individual tasks.

Though, not all variations of the Transformer architecture added additional performance. This project found that using relative positional encoding, i.e., clipping the positional encoding values beyond a certain distance in both forward and backward directions, did not increase model or individual task performance. As seen in Table 4, which contains a summary of task and RN accuracies across a wide gamut of maximum distance options, every task benefitted from a longer maximum distance value. These results disprove the theory that certain functional harmony tasks would benefit from only being able to see information near to the current time step. Relative positional encoding is not additive in this specific model architecture, though further experimentation and research could be done in this area beyond what was accomplished in this work.

	Avg. Chord Changes	**Avg. Unique Keys**	**Avg. Chord Duration (Measures)**
Validation Dataset Predictions (a)	589.2	7.1	0.9
Validation Dataset Actual (b)	306.4	5.3	1.8
Ratio (a/b)	1.9	1.3	0.5

Figure 43. A comparison of the predictions and actual average number of chord changes, unique keys, and chord durations of pieces in the validation dataset.

Micchi et al. noticed that their model predicted chord changes much more frequently than human annotators, attributing this to the granularity of the dataset itself. This trend was observed in this project's model's predictions as well. Figure 43 reports on the predicted and actual average number of chord changes per piece, unique keys per piece, and chord durations per piece in the validation dataset. The model predicts that the chord changes 1.9x more frequently than a human analyst believes it does, which signals there is further opportunity in research focused on including musical phrasing in the prediction of chord changes to ameliorate high frequency chord changes.

The research conducted during this project explored the application of multi-headed self-attention models to automatic harmonic analysis, proving its validity and surfacing the areas of further work. This project's efforts also sought to provide a better understanding of classical harmony at large and surface insights that could lead to better modelling in future research. It surfaced the model's ability to predict the most prominent chord variations, as well as its inability at correctly identifying more esoteric harmonic progressions.

5.2 Limitations of dataset, possible improvements

The generalisation of this project's model was hindered by class imbalance in the dataset and its smaller size relative to the larger datasets of other common machine learning tasks. In 2018, Popel and Bojar found that a Transformer translation model's BLEU score improved by 4% when trained on 58M sentences compared to 16M. (Popel & Bojar, 2018) Certain steps were taken to reduce overfitting, such as using the F1 score as an early stopping criterion that penalizes models that overfit to the majority class. Label smoothing was also implemented in the categorical cross entropy loss function to reduce the over confidence of the model. The model likely could have benefitted from implementing some sort of class weighting when defining the parameters of the loss function, but the dataset's limited size was not well-suited for this type of stratified regularisation approach.

5.2.1 Lack of rhythmic augmentation

As described in the methods section, each piece in the dataset was transposed as many times as possible while avoiding excessive numbers of accidentals in the pieces, e.g., F♭♭. This was done with the aim to not only increase the dataset's size but vary the distribution of key centres to promote generalisation on unseen data. This method of augmentation injects diversity into the pitches and subsequent chords of the dataset but does not introduce any variability in the time domain. López et al published a paper on Roman Numeral analysis after this project was already underway, where, in addition to building their own bespoke models that improve upon Micchi et al's work, they introduced 3 additional data augmentation techniques. Among these was **syncopation**, where a typically weak beat is instead strongly accented. They modified the scores so that the highest note of a chord is played first, followed by the rest of the notes, with the goal to occasionally shift the onset of the bass from the onset of the Roman numeral label. (López et al., 2021) This type of rhythmic augmentation would have been a valuable addition to this project's dataset, and likely would have improved the model's overall performance by avoiding overfitting on rhythmic changes specific to the dataset's limited set of composers.

5.3 More efficient hyperparameter searching

Optimizing the model's many hyperparameters proved to be a daunting task as the codebase and model structures were very custom and did not work well with common hyperparameter techniques like Bayesian optimization; this is due to the complex and bespoke nature of the RN modelling task. As such, a naïve random hyperparameter grid search was executed using the argparse Python module. During the randomized grid search, as the number of hyperparameters increased to nearly 40, generating the cartesian product of all hyperparameter options proved to be very memory intensive and limited the amount of experimentation that could be done at one time to find optimal model configurations. Further investigation in to, and the development of more efficient hyperparameter refinement methodologies for the RN classification task is recommended.

6. Evaluation Reflections and Conclusion

6.1 Overview

In this section, the project's objectives are evaluated and discussed, and the limitations experienced with hardware are summarised. The further areas of research and development identified during the research process are presented, and their motivations described. The section concludes with a personal reflection of the entire project's experience.

6.2 Choice of Research Questions and Objectives

This project's proposal initially included a second research question concerned with the ability to align audio data with the score data to provide additional signal-based data to make RN predictions. This task in combination with the encoder architecture implementation proved too daunting for this project's scope and was subsequently discarded. The objectives enumerated in this report's introductory chapter proved to be daunting on their own; this validated the decision in paring down this project's scope.

Further developing Micchi et al.'s functional harmony repository was more challenging than expected. The template Transformer codes utilised to develop this project's model used different software versions, specifically TensorFlow (Abadi et al., 2016). These issues introduced compatibility issues and subsequently required time-consuming efforts to integrate the Encoder model code into Micchi et al.'s model training and data encoding pipeline. However, developing this project's models within their existing repository made it relatively painless to compare the results of the two models in addition to recreating their results.

6.3 Limitations of the Project

Long model training times consumed a lot of this project's time. Models similar in architecture to the Transformer contain large number of learnable weights and subsequently require large amounts of VRAM to train. A graphics card with 8 GB of VRAM was used for this project, and while convenient for training locally as opposed to in a cloud environment like Google Colaboratory, a local GPU prevented the exploration of even larger models. An example of this in this in the project's model architecture would be the limitation in the number of encoder layers; initial grid searches would fail due to memory exhaustion errors when more than one task's encoder stack would contain more than 5 layers. This in turn created more development work to handle for these failures and continue training the next model. More VRAM would have allowed for larger models, but also allow all sizes of models to be trained with larger batches of input sequences, which could affect performance.

6.4 Future Work

While this project was able to complete the established objectives successfully, the research process yielded areas of further exploration and investigation. Among these is the implementation of an encoder / decoder model akin to N. Wu et al who leveraged this design from the Transformer paper, but in the context of time series forecasting. This is an avenue for exploration, with the potential benefits being the decoder's additional sub-layer that applies self-attention mechanisms over the upstream encoder output itself to further learn sequential dependencies. (N. Wu et al., 2020)

Applying rhythmic augmentation to the dataset has proven to increase RN classification performance as was seen in López et al.'s AugmentedNet. (López et al., 2021) This further augmentation combined with a self-attention based model could achieve state of the art results. López et al. also generated synthetic training samples via pitch contouring and via reducing the bass note's duration, which is also worthy of exploration within the context of this project's models.

Micchi et al.'s research indicated that encoding input piano rolls using pitch spelling as opposed to chromatic representations yielded more performant models. This was also true for including the bass notes as their own features concatenated to the piano roll data. Their repository contains support for six different types of input encoding not explored during this project, which may have surfaced different findings than above if further experimented with.

6.5 Reflection

From a personal perspective, as a musician myself, this project truly helped to enrich my knowledge of music theory and significantly strengthened my passion for music in the process. The model architecture needed for a task as nuanced as Roman numeral classification in turn required that I invest a great deal of time improving my knowledge of the TensorFlow framework. The academic research and development process is something that I was new to. This project facilitated a substantial increase to my work ethic, as well as my project management skills which are both directly translatable to industry. This project has further motivated me to pursue larger-scale individual projects outside of academia in the future.

7. Glossary

The below glossary contains definitions from Laitz's *The Complete Musician*, a well-known reference for music theory concepts. (Laitz, 2012)

Accidental: Symbols written before a note to raise or lower its pitch. Accidentals apply to pitches, not pitch classes, and are cancelled by bar lines.

Augmented: A triad that contains two major thirds, spanning an augmented fifth, which is therefore dissonant.

Chord: Any combination of two or more pitches sounding simultaneously.

Chromatic: Any pitch that lies outside the diatonic system. Also two pitches of the same letter name a half step apart (such as C and C#)

Circle of Fifths: The arrangement of the tonics of the 12 major or minor keys by ascending or descending perfect fifths, making a closed circle.

Diatonic: Literally "through the tones," meaning the collection of seven pitches in a major or minor scale with each letter name represented once, as distinct from chromatic pitches of the same letter name.

Diminished: A triad containing two minor thirds, spanning a diminished fifth, which is therefore a dissonant triad.

Diminished Interval: The smallest quality of interval. Perfect intervals and minor intervals become diminished when their size is decreased by one half step.

Dissonant Intervals: Unstable intervals, including the second, fourth (can be consonant), and seventh, their compound versions, and all diminished and augmented intervals.

Dominant: Scale degree 5, or the triad built on scale degree 5.

Enharmonic: Two notes that differ in name but not in pitch, such as C♯ and D♭.

Figured Bass: A shorthand notation used between 1600 and 1800 to describe the intervals above the bass note, also called thoroughbass. Figured bass always contains a bass note and a figure (unless the figure is purposely omitted, indicating a root-position triad).

Interval: The musical distance between two pitches.

Inversion: A chord with any chord tone other than the root in the bass is said to be in inversion.

Key: A musical composition based on a certain scale is said to be in that key.

Leading Tone: Scale degree 7 (raised 7 in the minor mode), a half step below tonic.s

Major/Minor: A category of intervals including the second, third, sixth, and seventh, or a possible quality for such intervals.

Major Mode: The scale organized with in the following steps: W–W–H–W–W–W–H

Minor Mode: A seven-pitch collection characterized by a minor third above 1.

Modulation: A large tonicization, which occupies an entire section of a piece, usually confirmed by a strong cadence in the new key.

Natural Minor Scale: One of the three forms of the minor scale, using lowered 6 and lowered 7. Minor key sig- natures refer to the natural minor scale.

Ornamentation: How a melody is varied or embellished in a score or performance.

Pitch: (1) A frequency generated by a vibrating body. (2) A musical tone or note.

Perfect Interval: A category of intervals including the unison, fourth, fifth, and octave.

Quality: The type of interval or chord (e.g., major, mi- nor, augmented, diminished).

Reduction: The process of removing neighbouring tones, consonant leaps, passing tones, etc., to reveal the underlying contrapuntal framework.

Rhythm: Durations of pitches and silences.

Root: The lowest pitch of a triad or seventh chord when stacked in thirds.

Scale Degrees: The numbering of the seven pitch classes in the diatonic scale, beginning with the tonic as 1. Each is marked by a caret.

Secondary Dominant: A dominant-functioning chord built on a scale degree other than 1, used for tonicization. A diminished triad cannot be tonicized.

Seventh Chord: A chord that in root position contains three stacked thirds, spanning the interval of a seventh. All seventh chords are dissonant, though to varying degrees.

Subdominant: Scale degree 4, or the triad built on scale degree 4.

Syncopation: A musical accent that occurs on a metrically unaccented beat or part of a beat.

Time Signature: Two numbers, one on top of the other, written at the very beginning of the piece. Top number = number of beats in a measure (simple) or number of beats in a measure once divided by three (compound); bottom number = note value of the beat.

Tonic: The gravitational centre of a set of pitches, called scale degree 1, or the triad built on scale degree 1.

Tonicization: Brief emphasis on a key other than tonic by means of one or more applied chords.

Triad: A chord comprised of three different pitches able to be stacked in thirds

8. References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., … Research, G. (2016). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*. https://doi.org/10.48550/arxiv.1603.04467

Chen, T.-P., & Su, L. (2018). *Functional Harmony Recognition of Symbolic Music Data with Multi-task Recurrent Neural Networks*. https://github.com/

Choi, K., Hawthorne, C., Simon, I., Dinculescu, M., & Engel, J. (2020). *Encoding Musical Style with Transformer Autoencoders*.

Cuthbert, C. S., Scott, M., & Ariza, C. (2010). *music21: A Toolkit for Computer-Aided Musicology and Symbolic Music Data*. http://ismir2010.ismir.net/proceedings/ismir2010-108.pdfhttp://hdl.handle.net/1721.1/84963http://creativecommons.org/licenses/by-nc-sa/3.0/

Davis, J., & Goadrich, M. (2006). *The Relationship Between Precision-Recall and ROC Curves*.

Devaney, J., Arthur, C., Condit-Schultz, N., & Nisula, K. (2015). Theme and variation encodings with roman numerals (TaVERn): A new data set for symbolic music analysis. *Proceedings of the 16th International Society for Music Information Retrieval Conference, ISMIR 2015*, 728–734. http://www.mturk.com/

Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2018). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *NAACL HLT 2019 - 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies - Proceedings of the Conference*, *1*, 4171–4186. https://doi.org/10.48550/arxiv.1810.04805

Hochreiter, S., & Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, *9*(8), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

Huang, G., Liu, Z., van der Maaten, L., & Weinberger, K. Q. (2016). *Densely Connected Convolutional Networks*. https://github.com/liuzhuang13/DenseNet.

Kingma, D. P., & Ba, J. L. (2014). Adam: A Method for Stochastic Optimization. *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*. https://arxiv.org/abs/1412.6980v9

Kiranyaz, S., Avci, O., Abdeljaber, O., Ince, T., Gabbouj, M., & Inman, D. J. (2020). *D Convolutional Neural Networks and Applications-A Survey*.

Laitz, S. G. (Steven G. (2012). *The complete musician : an integrated approach to tonal theory, analysis, and listening*. 875.

Liu, H.-M., & Yang, Y.-H. (2018). *Lead Sheet Generation and Arrangement by Conditional Generative Adversarial Network*. https://drive.google.com/open?

Liu, Z., Luo, S., Li, W., Lu, J., Wu, Y., Sun, S., Li, C., & Yang, L. (2020). *ConvTransformer: A Convolutional Transformer Network for Video Frame Synthesis*. http://arxiv.org/abs/2011.10185

López, N. N., Gotham, M. R. H., & Fujinaga, I. (2021). *AugmentedNet: A Roman Numeral Analysis Network with Synthetic Training Examples and Additional Tonal Tasks*. https://doi.org/10.5281/ZENODO.5624533

Ma, J., & Yarats, D. (2021). *On the Adequacy of Untuned Warmup for Adaptive Optimization*. www.aaai.org

Micchi, G., Gotham, M., & Giraud, M. (2020). Not All Roads Lead to Rome: Pitch Representation and Model Architecture for Automatic Harmonic Analysis. *Transactions of the International Society for Music Information Retrieval*, *3*(1), 42–54. https://doi.org/10.5334/tismir.45

Michel, P., Levy, O., & Neubig, G. (2019). Are Sixteen Heads Really Better than One? *NeurIPS*. https://github.com/neulab/compare-mt

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). *Efficient Estimation of Word Representations in Vector Space*. http://ronan.collobert.com/senna/

Nápoles López, N., Arthur, C., & Fujinaga, I. (2019). Key-Finding Based on a Hidden Markov Model and Key Profiles. *6th International Conference on Digital Libraries for Musicology*. https://doi.org/10.1145/3358664.3358675

Neuwirth, M., Harasim, D., Moss, F. C., & Rohrmeier, M. (2018). The Annotated Beethoven Corpus (ABC): A Dataset of Harmonic Analyses of All Beethoven String Quartets. *Frontiers in Digital Humanities*, *0*, 16. https://doi.org/10.3389/FDIGH.2018.00016

Popel, M., & Bojar, O. (2018). *Training Tips for the Transformer Model*. https://github.com/tensorflow/tensor2tensor

Rudbäck, & Niklas. (2020). *Circumscribing Tonality: Upper Secondary Music Students Learning the Circle of Fifths*. https://gupea.ub.gu.se/handle/2077/66147

Schuster, M., & Paliwal, K. K. (1997). Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, *45*(11), 2673–2681. https://doi.org/10.1109/78.650093

Shaw, P., Uszkoreit, J., & Vaswani, A. (2018). Self-Attention with Relative Position Representations. *NAACL HLT 2018 - 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies - Proceedings of the Conference*, *2*, 464–468. https://doi.org/10.18653/v1/n18-2074

Tymoczko, D., Gotham, M., Cuthbert, M. S., & Ariza, C. (2019). *The RomanText Format: A Flexible and Standard Method for Representing Roman Numerial Analyses*.

Vaswani, A., Brain, G., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). *Attention Is All You Need*.

Wu, K., Peng, H., Chen, M., Fu, J., & Chao, H. (2021). *Rethinking and Improving Relative Position Encoding for Vision Transformer*. http://arxiv.org/abs/2107.14222

Wu, N., Green, B., Ben, X., & O'banion, S. (2020). *Deep Transformer Models for Time Series Forecasting: The Influenza Prevalence Case*.

9. Appendix

9.1 The Circle of Fifths

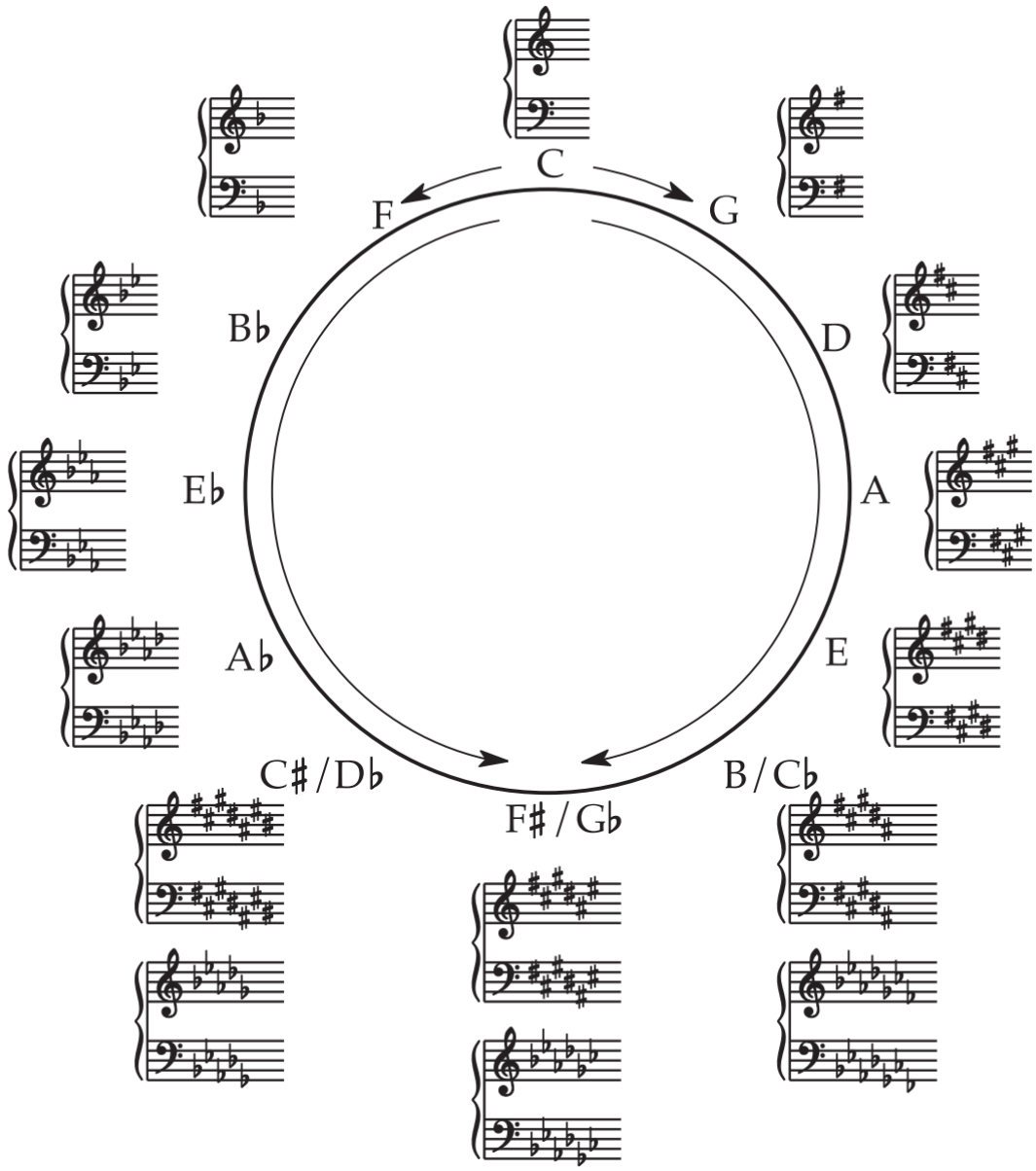

Figure 44. The circle of fifths. This diagram from The Complete Musician illustrates how keys one fifth apart from each other are harmonically related. Keys opposite one another on the circle are very harmonically dissimilar from each other and sound jarring one after another.

9.2 Code Related to RN Model Construction

The full repository of this project's code can be found at this OneDrive link.

```
class EncoderLayer(tf.keras.layers.Layer):
    """
    Used https://www.tensorflow.org/text/tutorials/transformer as a template, heavily modified from source.
    Individual layer in Encoder stack. Multi-headed self-attention component is contained within it.
    """

    def __init__(self, d_model, num_heads, dff, layer_num, relative, max_distance, rate=0.1, **kwargs):
        """

        Args:
            d_model (int): Number of features in the input vector. Used when calculating MHA depth.
            num_heads (int): Number of attention heads to use.
            dff (int): Dimension of Feed Forward layer after attention block.
            layer_num (int): Numeric identifier for each encoder layer in the encoder stack.
            relative (bool): Boolean flag indicating whether or not to use relative or absolute positional
                encoding within the attention block.
            max_distance (int): Maximum distance in both forward and backward directions that relative
                positional encoding will utilise when clipping values.
            rate (float): Dropout rate between 0 and 1 that is used in the dropout layer after calculating
                attention weights, and after the feed forward layer.
        """
        super(EncoderLayer, self).__init__(**kwargs)

        self.num_heads = num_heads
        self.dff = dff
        self.rate = rate
        self.d_model = d_model
        self.layer_num = layer_num
        self.relative = relative
        self.max_distance = max_distance

        # instantiate multi-head attention block
        self.mha = MultiHeadAttention(
            self.d_model,
            self.num_heads,
            self.layer_num,
            relative=relative,
            max_distance=self.max_distance,
            name=f"{self.name}_mha_{self.layer_num}",
            rate=self.rate,
        )
        # construct dense layer with normalization and configure dropout layers
        self.ffn = point_wise_feed_forward_network(self.d_model, self.dff, layer_num, name=self.name)
        self.layernorm1 = tf.keras.layers.LayerNormalization(
            epsilon=1e-5,
            name=f"{self.name}_layernorm1_{self.layer_num}"
        )
        self.layernorm2 = tf.keras.layers.LayerNormalization(
            epsilon=1e-5,
            name=f"{self.name}_layernorm2_{self.layer_num}"
        )
        self.dropout1 = tf.keras.layers.Dropout(self.rate, name=f"{self.name}_dropout1_{self.layer_num}")
        self.dropout2 = tf.keras.layers.Dropout(self.rate, name=f"{self.name}_dropout2_{self.layer_num}")

    def get_config(self):
        """
        Necessary to override this method so the model can be saved and reloaded.
        https://stackoverflow.com/questions/58678836/notimplementederror-layers-with-arguments-in
        -init-must-override-get-conf
        """
        config = super().get_config().copy()
        config.update(
            {
                "layer_num": self.layer_num,
```

```
                "d_model": self.d_model,
                "dff": self.dff,
                "num_heads": self.num_heads,
                "rate": self.rate,
                "relative": self.relative,
            }
        )
        return config

    def call(self, x, is_training=False, mask=None):
        """Method that processes each batch through the model architecture."""
        # get attention scores
        attn_output, _ = self.mha(x, x, x, mask, is_training)  # (batch_size, input_seq_len, d_model)
        attn_output = self.dropout1(attn_output, training=is_training)

        # the MHA output is added to the original input. This is a residual connection.
        out1 = self.layernorm1(x + attn_output, training=is_training)  # (batch_size, input_seq_len, d_model)

        # ffn a few linear layers with ReLU
        ffn_output = self.ffn(out1)  # batch_size, input_seq_len, d_model)
        ffn_output = self.dropout2(ffn_output, training=is_training)

        # another residual connection that allows gradients to flow through network continuously
        out2 = self.layernorm2(out1 + ffn_output, training=is_training)
        return out2

class Encoder(tf.keras.layers.Layer):
    """Uses https://www.tensorflow.org/text/tutorials/transformer as a template, but heavily modified."""

    def __init__(
        self,
        num_layers,
        d_model,
        num_heads,
        dff,
        maximum_position_encoding,
        rate=0.1,
        relative=False,
        max_distance=None,
        **kwargs,
    ):
        """
        Construct the larger encoder stack with constituent Encoder layers. During the instantiation process
        an individual EncoderLayer object is instantiated for the num_layers provided as a class argument.

        When this object is called, the input batch is modified by the positional encoding function to
        encode sequential order. This is then passed through each encoder layer, its outputs are updated with
        attention scores and its output returned.
        Args:
            num_layers (int): Number of encoder layers to construct within the larger encoder stack.
            d_model (int): Number of input features.
            num_heads (int): Number of heads to use when calculating self-attention scores.
                This must be true: d_model % num_heads == 0
            dff (int): Dimension of Feed Forward layer after attention block in each encoder layer.
            maximum_position_encoding (int): Determines the size number of time steps of the positional
                encoding array.
            rate (float): Dropout rate between 0 and 1 that is used in the dropout layer after calculating
                attention weights, and after the feed forward layer.
            relative (bool): Boolean flag indicating whether or not to use relative or absolute positional
                encoding within the attention block.
            max_distance (int): Maximum distance in both forward and backward directions that relative
                positional encoding will utilise when clipping values.
            **kwargs:
        """
        super(Encoder, self).__init__(**kwargs)

        self.d_model = d_model
        self.num_layers = num_layers
        self.num_heads = num_heads
        self.max_pos_encoding = maximum_position_encoding
        self.rate = rate
```

```
        self.dff = dff
        self.relative = relative
        self.max_distance = max_distance

        # get positional encoding vector
        self.pos_encoding = positional_encoding(self.max_pos_encoding, self.d_model)

        self.enc_layers = [
            EncoderLayer(
                self.d_model,
                self.num_heads,
                self.dff,
                relative=self.relative,
                max_distance=self.max_distance,
                rate=self.rate,
                layer_num=n,
                name=f"{self.name}_enc_{n}",
            )
            for n in range(self.num_layers)
        ]  # instantiate all of the encoder layers in the encoder stack for this task

        self.dropout = tf.keras.layers.Dropout(self.rate, name=f"{self.name}_dropout")

    def get_config(self):
        """# https://stackoverflow.com/questions/58678836/notimplementederror-layers-
        with-arguments-in-init-must-override-get-conf"""
        config = super().get_config().copy()
        config.update(
            {
                "num_layers": self.num_layers,
                "d_model": self.d_model,
                "dff": self.dff,
                "num_heads": self.num_heads,
                "rate": self.rate,
                "max_distance": self.max_distance,
                "relative": self.relative,
                "maximum_position_encoding": self.max_pos_encoding,
            }
        )
        return config

    def call(self, x, is_training=False, mask=None):
        """Add positional encoding vector, then pass through each encoder layer."""
        seq_len = tf.shape(x)[1]

        # (batch_size, time_steps, pitches), eg (128, 160, 70)
        x *= tf.math.sqrt(tf.cast(self.d_model, tf.float32))
        x += self.pos_encoding[:, :seq_len, :]
        x = self.dropout(x, training=is_training)

        for i in range(self.num_layers):
            x = self.enc_layers[i](x, is_training=is_training, mask=mask)
        return x  # (batch_size, input_seq_len, d_model)


def build_task_model(x, enc, num_classes, task, dff_2,
                     dropout_rate, pool_type, is_training=False, concat_layers=[]):
    """
    Downsamples input data using provided method, passes batch through instantiated encoder applying
    self attention. Optionally concatenates the outputs of other task models to further inform the decision
    of the current task. Applies regularization layers like Dropout and LayerNorm to prevent overfitting
    and gradient explosion. Returns prediction probabilities via softmax activation function.
    Args:
        x: Input batch
        enc (encoder object): Encoder object for an individual Roman numeral task.
        num_classes (int): Number of output classes for the task, determines the shape of the softmax layer.
        task (str): Roman numeral task. e.g., Key, Degree 2
        dff_2 (int): Dimension of the hidden layer between the Encoder stack and output layer.
        dropout_rate (float): Dropout used in between Encoder and hidden layer.
        pool_type (str): Determines the downsampling method prior to going through the encoder stack.
        is_training (bool): Indicates whether or not to use random weights in regularization layers.
```

```
        concat_layers (bool): List of output layers from previous models to include in the
            predictions of the current layer.

    Returns:
        Tensor of dimension (batch_size, num_classes)
    """
    # (batch_size, time_steps, pitches)
    x = get_pool(pool_type=pool_type, task=task)(x)  # (batch_size, input_seq_len)

    if concat_layers:  # add outputs from previous layers if applicable
        for idx, conc_layer in enumerate(concat_layers):
            # optionally add key predictions to degree inputs
            x = Concatenate(name=f"{task}_{idx}_concat")([x, conc_layer])

    x = enc(x, is_training=is_training, mask=None)  # (batch_size, seq_len, pitches)
    x = LayerNormalization(name=f"{task}_mt_ln_1")(x, training=is_training)
    x = Dropout(rate=dropout_rate, name=f"{task}_mt_dropout_1")(x, training=is_training)
    x = Dense(dff_2, activation="relu", name=f"{task}_mt_f1")(x)
    x = LayerNormalization(name=f"{task}_mt_ln_2")(x, training=is_training)
    output = Dense(num_classes, activation="softmax", name=f"{task}_mt_f2")(x)
    return output


def get_encoder_stacks(x, encoder_dict, input_type, dff_2,
                       dropout_rate, pool_type, key_chain, quality_chain, is_training=False):
    """
    Construct the model around the instantiated encoder stacks and relate the task models together.
    Returns the predictions in an array containing the output matrices for each batch per task.

    Args:
        x (Tensor): Input batch
        encoder_dict (dict): Dictionary of encoder objects containing layers of multitask
            attention mechanisms. Dict keys are the individual roman numeral tasks.
        input_type (str): Type of input representation, determines the prediction labels.
        dff_2 (int): Dimension of the hidden layer between the Encoder stack and output layer.
        dropout_rate (float): Dropout used in between task's Encoder and hidden layer.
        pool_type (str): Determines the downsampling method prior to going through the encoder stack.
        key_chain (bool): Boolean indicating if predictions from key task should be passed on to
            all other tasks.
        quality_chain (bool): Boolean indicating if quality predictions are to be concatenated to
            input prior to the inversion and degree task encoders.
        is_training (bool): Indicates if random weights are to be used in regularization layers.
            Defaults to False.

    Returns:
        List of output matrices in order [key, degree_1, degree_2, quality, inversion, root]
    """
    classes_key = 30 if input_type.startswith("spelling") else 24  # Major keys: 0-11, Minor keys: 12-23
    # 7 degrees * 3: regular, diminished, augmented
    classes_degree = 21
    # the twelve notes without enharmonic duplicates
    classes_root = 35 if input_type.startswith("spelling") else 12
    classes_quality = 12  # ['M', 'm', 'd', 'a', 'M7', 'm7', 'D7', 'd7', 'h7', 'Gr+6', 'It+6', 'Fr+6']
    classes_inversion = 4  # root position, 1st, 2nd, and 3rd inversion (the last only for seventh chords)

    # get key prediction
    key = build_task_model(
        x,
        enc=encoder_dict["key"],
        num_classes=classes_key,
        task="key",
        dff_2=dff_2,
        dropout_rate=dropout_rate,
        pool_type=pool_type,
        is_training=is_training,
    )
    concat_layers = [key] if key_chain else []
    root = build_task_model(  # root informed by key predictions
        x,
        enc=encoder_dict["root"],
        num_classes=classes_root,
```

```
        task="root",
        dff_2=dff_2,
        dropout_rate=dropout_rate,
        pool_type=pool_type,
        is_training=is_training,
        concat_layers=concat_layers,
    )
    quality = build_task_model(  # quality informed by key predictions
        x,
        enc=encoder_dict["quality"],
        num_classes=classes_quality,
        task="quality",
        dff_2=dff_2,
        dropout_rate=dropout_rate,
        pool_type=pool_type,
        is_training=is_training,
        concat_layers=concat_layers,
    )
    concat_layers = concat_layers + [root] if key_chain else concat_layers
    concat_layers = concat_layers + [quality] if quality_chain else concat_layers
    degree_1 = build_task_model(  # degree 1 informed by key, quality, and root predictions
        x,
        enc=encoder_dict["dg1"],
        num_classes=classes_degree,
        task="dg1",
        dff_2=dff_2,
        dropout_rate=dropout_rate,
        pool_type=pool_type,
        concat_layers=concat_layers,
        is_training=is_training,
    )
    degree_2 = build_task_model(  # degree 2 informed by key, quality, and root predictions
        x,
        enc=encoder_dict["dg2"],
        num_classes=classes_degree,
        task="dg2",
        dff_2=dff_2,
        dropout_rate=dropout_rate,
        pool_type=pool_type,
        concat_layers=concat_layers,
        is_training=is_training,
    ) # should be informed by key and quality because only 7th notes have certain inversions
    inversion = build_task_model(
        x,
        enc=encoder_dict["inv"],
        num_classes=classes_inversion,
        task="inversion",
        dff_2=dff_2,
        dropout_rate=dropout_rate,
        pool_type=pool_type,
        is_training=is_training,
        concat_layers=concat_layers,
    )
    return [key, degree_1, degree_2, quality, inversion, root]


def create_rn_model(
    d_model,
    dff_1,
    max_pos_encoding,
    input_type,
    dff_2,
    pool_type,
    key_chain,
    quality_chain,
    max_distance,
    h_params,
    is_training=False,
    dropout_rate=0.15,
):
    """
```

```
    Instantiates an Encoder Block for each of the RN component tasks. Model input dimensions can vary
    if key_chain or quality_chain are True. Passes the instantiated Encoder blocks to a function that
    generates predictions for each task and returns them in a list.
    Args:
        d_model (int): Number of features in the input vector. Used when calculating MHA depth.
        dff_1 (int): Dimension of Feed Forward layer after attention block.
        max_pos_encoding (int): Indicates the size of the positional encoding matrix.
            This project used 640 (160 * 4).
        input_type (str): Type of input encoding used for model training. This project uses
            'spelling_complete_cut'.
        dff_2 (int): Dimension of hidden Feed Forward layer prior to softmax activation layer.
        pool_type (str): Type of downsampling method used on the input sequence to reduce
            dimensionality to match output chord frequency.
        key_chain (bool): Boolean flag indicating if the outputs from the Key task are to be
            concatenated to other task inputs.
        quality_chain (bool): Boolean flag indicating if the outputs from the Quality task are
            to be concatenated to other downstream tasks.
        max_distance (int): Maximum distance in both forward and backward directions that
            relative positional encoding will utilise when clipping values.
        h_params (dict): Dictionary of hyperparameters passed via the command line.
        is_training (bool): Indicates if the model is in a training stage or not. Useful for
            dropout layers that use random components.
        dropout_rate (float): Dropout rate to use in all model dropout layers.

    Returns:
        keras.Model object suitable for training the multi-task RN problem.
    """
    notes_input = Input(shape=(None, d_model), name="piano_roll_input")  # (time_steps, pitches)
    empty_mask = Input(shape=(None, 1), name="mask_input")  # (time_steps / 4, 1)
    blank_1 = Input(shape=(None,), name="_1", dtype=tf.string)  # for filename, not needed
    blank_2 = Input(shape=(None,), name="_2")  # for transposition, not needed
    blank_3 = Input(shape=(None,), name="_3")  # for start time, not needed

    dg_quality = d_model + 30 if key_chain else d_model  # chain key predictions 100
    dg_root = d_model + 30 if key_chain else d_model  # 100

    # add key and root to chain (root informed by key, too)  135 or 70
    dg_d_model_inv = d_model + 30 + 35 if key_chain else d_model
    # add quality outputs if also chained 147 or 82
    dg_d_model_inv = dg_d_model_inv + 12 if quality_chain else dg_d_model_inv

    key_enc = Encoder(  # which key the chords relate to
        num_layers=h_params[hp_config.HP_KEY_NEL],
        d_model=d_model,
        num_heads=h_params[hp_config.HP_KEY_NAH],
        dff=dff_1,
        maximum_position_encoding=max_pos_encoding,
        rate=h_params[hp_config.HP_KEY_DROPOUT_RATE],
        relative=bool(h_params[hp_config.HP_KEY_REL]),
        max_distance=max_distance if bool(h_params[hp_config.HP_KEY_REL]) else None,
        name="key_encoder",
    )

    root_enc = Encoder(  # root of the chord, can differ from bass note if inverted
        num_layers=h_params[hp_config.HP_ROOT_NEL],
        d_model=dg_root,
        num_heads=h_params[hp_config.HP_ROOT_NAH],
        dff=dff_1,
        maximum_position_encoding=max_pos_encoding,
        rate=dropout_rate,
        relative=bool(h_params[hp_config.HP_ROOT_REL]),
        max_distance=max_distance if bool(h_params[hp_config.HP_ROOT_REL]) else None,
        name="root_encoder",
    )

    quality_enc = Encoder(  # chord quality, major minor etc
        num_layers=h_params[hp_config.HP_QUALITY_NEL],
        d_model=dg_quality,
        num_heads=h_params[hp_config.HP_QUALITY_NAH],
        dff=dff_1,
        maximum_position_encoding=max_pos_encoding,
```

```
            rate=dropout_rate,
            relative=bool(h_params[hp_config.HP_QUALITY_REL]),
            max_distance=max_distance if bool(h_params[hp_config.HP_QUALITY_REL]) else None,
            name="quality_encoder",
        )

        dg1_enc = Encoder(  # secondary dominants
            num_layers=h_params[hp_config.HP_DG1_NEL],
            d_model=dg_d_model_inv,
            num_heads=h_params[hp_config.HP_DG1_NAH],
            dff=dff_1,
            maximum_position_encoding=max_pos_encoding,
            rate=dropout_rate,
            relative=bool(h_params[hp_config.HP_DG1_REL]),
            max_distance=max_distance if bool(h_params[hp_config.HP_DG1_REL]) else None,
            name="dg1_encoder",
        )

        dg2_enc = Encoder(  # chord dg relative to dg1
            num_layers=h_params[hp_config.HP_DG2_NEL],
            d_model=dg_d_model_inv,
            num_heads=h_params[hp_config.HP_DG2_NAH],
            dff=dff_1,
            maximum_position_encoding=max_pos_encoding,
            rate=dropout_rate,
            relative=bool(h_params[hp_config.HP_DG2_REL]),
            max_distance=max_distance if bool(h_params[hp_config.HP_DG2_REL]) else None,
            name="dg2_encoder",
        )

        inv_enc = Encoder(  # chord inversion, 4 possible
            num_layers=h_params[hp_config.HP_INVERSION_NEL],
            d_model=dg_d_model_inv,
            num_heads=h_params[hp_config.HP_INVERSION_NAH],
            dff=dff_1,
            maximum_position_encoding=max_pos_encoding,
            rate=dropout_rate,
            relative=bool(h_params[hp_config.HP_INVERSION_REL]),
            max_distance=max_distance if bool(h_params[hp_config.HP_INVERSION_REL]) else None,
            name="inv_encoder",
        )

        encs_dict = {  # save each encoder to a dictionary to pass to the model building function
            "dg1": dg1_enc,
            "dg2": dg2_enc,
            "quality": quality_enc,
            "inv": inv_enc,
            "key": key_enc,
            "root": root_enc,
        }

        y = get_encoder_stacks(
            notes_input,
            encs_dict,
            input_type=input_type,
            dff_2=dff_2,
            dropout_rate=dropout_rate,
            pool_type=pool_type,
            key_chain=key_chain,
            quality_chain=quality_chain,
            is_training=is_training,
        )
        model = Model(inputs=[notes_input, empty_mask, blank_1, blank_2, blank_3], outputs=y)
        return model

    class MultiHeadAttention(tf.keras.layers.Layer):
        """
        Multi-headed attention layer based on TensorFlow's Transformer tutorial:
        https://www.tensorflow.org/text/tutorials/transformer

        Uses and combines components relative positional encoding components from from:
```

```
https://github.com/tensorflow/tensor2tensor/blob/5623deb79cfcd28f8f8c5463b58b5bd76a81fd0d/
tensor2tensor/layers/common_attention.py

Module in a Transformer network that computes the attention weights
for the input and produces an output vector with encoded information
on how each token should attend to all other tokens in the sequence.
"""

def __init__(self, d_model, num_heads, layer_num, relative, rate, max_distance, **kwargs):
    """
    Args:
        d_model (int): Number of features in the input vector. Used when calculating MHA depth.
        num_heads (int): Number of attention heads to use.
        layer_num (int): Numeric identifier for each encoder layer in the encoder stack.
        relative (bool): Boolean flag indicating whether or not to use relative or absolute positional
        encoding within the attention block.
        rate (float): Dropout rate between 0 and 1 that is used in the dropout layer after calculating
        attention weights, and after the feed forward layer.
        max_distance (int): Maximum distance in both forward and backward directions that relative
        positional encoding will utilise when clipping values.
        **kwargs:
    """
    super(MultiHeadAttention, self).__init__(**kwargs)

    self.num_heads = num_heads
    self.d_model = d_model
    self.layer_num = layer_num
    self.relative = relative
    self.max_distance = max_distance
    self.mha_name = f"{self.name}_{self.layer_num}"

    # necessary that the number of attention heads is a factor of number of features
    assert d_model % num_heads == 0

    self.depth = d_model // self.num_heads  # calculate depth
    self.dropout_rate = rate

    # initialise w matrices
    self.wq = tf.keras.layers.Dense(d_model, name=f"{self.name}_mha_wq_{layer_num}")
    self.wk = tf.keras.layers.Dense(d_model, name=f"{self.name}_mha_wk_{layer_num}")
    self.wv = tf.keras.layers.Dense(d_model, name=f"{self.name}_mha_wv_{layer_num}")

    # output layer for absolute positional encoding
    self.dense = tf.keras.layers.Dense(d_model, name=f"{self.name}_mha_dense_{layer_num}")

    # relative positional encoding layers
    self.dense_rpe = Dense(self.d_model, name=f"dense_rel_pe_{self.layer_num}")  # d_model should be 70
    self.dropout = Dropout(self.dropout_rate)
    self.dense_output = Dense(self.d_model, name=f"dense_O_{self.layer_num}")
    self.layer_norm = LayerNormalization()

def call(self, v, k, q, mask, is_training=False):
    """
    Method that calculates self attention scores for each input batch.

    The boolean class attribute `self.relative` indicates whether or not to
    use relative positional encoding when calculating attention scores. If equal to False,
    absolute attention is used as it was implemented in the Transformer model.
    If equal to True, relative attention scores are calculated using relative positional
    encoding. Max distance is provided as an instantiation argument of the class.

    Args:
        v (tensor): value must be (batch_size, seq_len_v, depth_v)
        k (tensor): key must be (batch_size, seq_len_k, depth)            k:
        q (tensor): query must be (batch_size, seq_len_q, depth)
        mask (Float tensor): Unused parameter in this application.
        is_training (bool):

    Returns:
        attention_scores, attention_weights
    """
```

```
        batch_size = tf.shape(q)[0]

        Q = self.wq(q)  # (batch_size, seq_len, d_model)
        K = self.wk(k)  # (batch_size, seq_len, d_model)
        V = self.wv(v)  # (batch_size, seq_len, d_model)

        q = self.split_heads(Q)  # (batch_size, num_heads, seq_len_q, depth)
        k = self.split_heads(K)  # (batch_size, num_heads, seq_len_k, depth)
        v = self.split_heads(V)

        if not self.relative:
            # split q, k, v in to N vectors before applying self attention
            # the split vectors then go through the same attention process individually

            # (batch_size, num_heads, seq_len_v, depth)
            # scaled_attention.shape == (batch_size, num_heads, seq_len_q, depth)
            # attention_weights.shape == (batch_size, num_heads, seq_len_q, seq_len_k)
            scaled_attention, attention_weights = scaled_dot_product_attention(q, k, v, mask)
            scaled_attention = tf.transpose(scaled_attention, perm=[0, 2, 1, 3])
            concat_attention = tf.reshape(
                scaled_attention,
                (batch_size, -1, self.d_model)
            )  # TensorShape([32, 80, 70]) concat different attention heads
            output = self.dense(concat_attention)  # (batch_size, seq_len_q, d_model)

        else:
            # if relative positional encoding
            total_key_depth = K.get_shape().as_list()[-1]
            total_value_depth = V.get_shape().as_list()[-1]

            q = split_heads(Q, self.num_heads)
            key_depth_per_head = total_key_depth // self.num_heads
            q *= key_depth_per_head**-0.5
            x, attention_weights = dot_product_attention_relative(
                q,
                k,
                v,
                bias=None,
                max_relative_position=self.max_distance,
                dropout_rate=self.dropout_rate,
                image_shapes=None,
                save_weights_to=None,
                make_image_summary=False,
                cache=False,
                allow_memory=False,
                hard_attention_k=0,
                gumbel_noise_weight=0.0,
                name=self.mha_name,
            )
            x = combine_heads(x)
            x.set_shape(x.shape.as_list()[:-1] + [total_value_depth])
            output = self.dense_output(x)
        return output, attention_weights
```

9.3 Grid Search and Training Code

```
# GRID SEARCH CODE
if __name__ == '__main__':

    num_combos = 1
    # get number of unique combos to iterate through for progress bar
    for name, param in hp_config.HPARAMS_DICT.items():
        num_combos *= len(param.domain.values)

    possible_values_arr = [x.domain.values for x in hp_config.HPARAMS_DICT.values()]
    hparam_keys = [x for x in hp_config.HPARAMS_DICT.keys()]
    h_param_arr = list()
```

```
# create all unique combos of hyperparameters
with tqdm(total=num_combos) as progress_bar:
    for combo in itertools.product(*possible_values_arr):
        h_param_arr.append(dict(zip(hparam_keys, combo)))
        progress_bar.update(1)

# initialize grid search hparams to record in tensorboard
with tf.summary.create_file_writer("logs/hparam_tuning").as_default():
    hp.hparams_config(
        hparams=[
            hp_config.HP_D_MODEL,
            hp_config.HP_BATCH_SIZE,
            hp_config.HP_DROPOUT,
            hp_config.HP_D_FF_1,
            hp_config.HP_D_FF_2,
            hp_config.HP_LR,
            hp_config.HP_EPOCHS,
            hp_config.HP_POOL_TYPE,
            hp_config.HP_KEY_CHAIN,
            hp_config.HP_REL_POS_ENC,
            hp_config.HP_WARM_UP_STEPS,
            hp_config.HP_MAX_DISTANCE,
            hp_config.HP_D_MODEL_MULT,
            hp_config.HP_MAX_POS_ENCODING,
            hp_config.HP_LR_SCHED,
            hp_config.HP_CHUNK_SIZE,
            hp_config.HP_DG1_NEL,
            hp_config.HP_DG2_NEL,
            hp_config.HP_QUALITY_NEL,
            hp_config.HP_INVERSION_NEL,
            hp_config.HP_KEY_NEL,
            hp_config.HP_ROOT_NEL,
            hp_config.HP_DG1_NAH,
            hp_config.HP_DG2_NAH,
            hp_config.HP_QUALITY_NAH,
            hp_config.HP_INVERSION_NAH,
            hp_config.HP_KEY_NAH,
            hp_config.HP_ROOT_NAH,
            hp_config.HP_DG1_REL,
            hp_config.HP_DG2_REL,
            hp_config.HP_QUALITY_REL,
            hp_config.HP_INVERSION_REL,
            hp_config.HP_KEY_REL,
            hp_config.HP_ROOT_REL,
            hp_config.HP_QUALITY_CHAIN,
            hp_config.HP_EPSILON,
            hp_config.HP_BETA_ONE,
            hp_config.HP_BETA_TWO,
            hp_config.HP_LABEL_SMOOTHING,
            hp_config.HP_KEY_DROPOUT_RATE

        ],
        metrics=[hp.Metric(hp_config.METRIC_KEY_ACCURACY, display_name="key"),
                 hp.Metric(hp_config.METRIC_DG1_ACCURACY, display_name="degree 1"),
                 hp.Metric(hp_config.METRIC_DG2_ACCURACY, display_name="degree 2"),
                 hp.Metric(hp_config.METRIC_QUALITY_ACCURACY, display_name="quality"),
                 hp.Metric(hp_config.METRIC_INV_ACCURACY, display_name="inversion"),
                 hp.Metric(hp_config.METRIC_ROOT_ACCURACY, display_name="root"),
                 hp.Metric(hp_config.METRIC_DEGREE_ACCURACY, display_name="degree"),
                 hp.Metric(hp_config.METRIC_SECONDARY_ACCURACY, display_name="secondary"),
                 hp.Metric(hp_config.METRIC_ROMAN_ACCURACY, display_name="roman"),
                 hp.Metric(hp_config.METRIC_ROMAN_INV_ACCURACY, display_name="roman + inv"),
                 hp.Metric(hp_config.METRIC_ROOT_COHERENCE_ACCURACY, display_name="root coherence"),
                 hp.Metric(hp_config.METRIC_D7_NO_INC_ACCURACY, display_name="d7 no inv")
                 ],
    )

for _ in tqdm(h_param_arr):
    combo = random.choice(h_param_arr)
    # positional encoding needs to be big enough for data
    if combo['max_pos_encoding'] < combo['chunk_size']:
```

```
            continue

        # execute training file for this specific hyperparameter combination
        args = f"-b {combo['batch_size']}  -e {combo['epochs']} " \
               f"-lr {combo['learning_rate']} " \
               f"-d {combo['dropout']} -dff1 {combo['dimension_ff_1']} -dff2 {combo['dimension_ff_2'] } " \
               f"-pt {combo['pool_type']} -mpe {combo['max_pos_encoding']} " \
               f"-dm {combo['d_model']} -kc {combo['key_chain']} -wu {combo['warm_up_steps']} " \
               f"-md {combo['max_distance']} " \
               f"-dmm {combo['d_model_mult']}  -ls {combo['lr_scheduler']} -r {combo['rel_pos_enc']} " \
               f"-cs {combo['chunk_size']} " \
               f"-dg1-nel {combo['dg1_nel']} -dg2-nel {combo['dg2_nel']} -q-nel {combo['quality_nel']} " \
               f"-inv-nel {combo['inversion_nel']} " \
               f"-k-nel {combo['key_nel']} -r-nel {combo['root_nel']} " \
               f"-dg1-nah {combo['dg1_nah']} -dg2-nah {combo['dg2_nah']} -q-nah {combo['quality_nah']} " \
               f"-inv-nah {combo['inversion_nah']} " \
               f"-k-nah {combo['key_nah']} -r-nah {combo['root_nah']} " \
               f"-dg1-rel {combo['dg1_rel']} -dg2-rel {combo['dg2_rel']} -q-rel {combo['quality_rel']} " \
               f"-inv-rel {combo['inversion_rel']} " \
               f"-k-rel {combo['key_rel']} -r-rel {combo['root_rel']} -qc {combo['quality_chain']} " \
               f"-lse {combo['label_smoothing_eps']} "\
               f"-b1 {combo['beta_1']} -b2 {combo['beta_2']} -eps {combo['epsilon']} " \
               f"-kd {combo['key_dropout']} "\
               f"--input 4"

        subprocess.call(f"../venv/Scripts/python encoder_train.py {args}")

# MODEL TRAINING CODE
def main(cli_args):
    """
    Train and save the model using hyperparameters passed via argparse.
    Args:
        cli_args: Argparse command line argument namespace containing hyperparameter values.

    Returns:
        None
    """
    print_cli_args(cli_args)
    h_params = create_hyperparams(cli_args)
    model_type, input_type = shared_config.MODELS[cli_args.model_idx],
shared_config.INPUT_TYPES[cli_args.input_idx]
    tfrecords_dir = os.path.join(shared_config.DATA_FOLDER, f'{h_params[hp_config.HP_CHUNK_SIZE]}_chunk')

    train_path, valid_path = setup_tfrecords_paths(tfrecords_dir, ["train", "valid"], input_type)
    model_folder, model_name = setup_model_paths(shared_config.EXPLORATORY, model_type, input_type)
    model_path = os.path.join(model_folder, model_name + ".h5")
    write_hparams_csv(cli_args, model_name)

    with tf.summary.create_file_writer("logs/hparam_tuning/" + model_name).as_default():
        hp.hparams(
            h_params,
            trial_id=model_name
        )  # record the values used in this trial

        # train model
        accuracies = train_encoder_model(
            h_params,
            train_path,
            valid_path,
            input_type,
            model_path,
            model_folder,
            model_name
        )
        register_metrics(accuracies)


def train_encoder_model(h_params, train_path, valid_path,
                        input_type, model_path, model_folder, model_name):
    """
    Args:
```

```
    h_params (dict): Dictionary of hyperparameters passed via the command line.
    train_path (str): Filepath containing training TFRecords.
    valid_path (str): Filepath containing validation TFRecords.
    input_type (str): Type of input encoding used for model training.
        This project uses 'spelling_complete_cut'.
    model_path (str): Filepath indicating where .h5 model file will be saved.
    model_folder (str): Directory location where TensorBoard runtime events and model will be saved.
    model_name (str): Name of the model to be saved.

Returns:
    (dict) Dictionary of training accuracies for each task, as well as overall RN accuracies.

"""
train_data = load_tfrecords_dataset(
    train_path,
    h_params[hp_config.HP_BATCH_SIZE],
    shared_config.SHUFFLE_BUFFER,
    input_type,
    h_params[hp_config.HP_CHUNK_SIZE],
    repeat=False,
)
valid_data = load_tfrecords_dataset(
    valid_path,
    h_params[hp_config.HP_BATCH_SIZE],
    1,
    input_type,
    h_params[hp_config.HP_CHUNK_SIZE],
    repeat=False
)
input_shape = hp_config.INPUT_TYPE2INPUT_SHAPE[input_type]
batch_size, num_features, seq_length = get_batch_attributes(train_data)
print_batch_attributes(batch_size, h_params, num_features, seq_length)

model = create_rn_model(
    d_model=input_shape,
    dff_1=h_params[hp_config.HP_D_FF_1],
    max_pos_encoding=h_params[hp_config.HP_MAX_POS_ENCODING],
    h_params=h_params,
    input_type=input_type,
    dff_2=h_params[hp_config.HP_D_FF_2],
    pool_type=h_params[hp_config.HP_POOL_TYPE],
    key_chain=h_params[hp_config.HP_KEY_CHAIN],
    quality_chain=h_params[hp_config.HP_QUALITY_CHAIN],
    max_distance=h_params[hp_config.HP_MAX_DISTANCE],
    is_training=True,
)
model.summary()
watch_metric = "val_loss"
callbacks = get_callbacks(h_params, model_folder, model_name, model_path, watch_metric)
lr_schedule = get_lr_scheduler(h_params)

opt = Adam(
    learning_rate=lr_schedule,
    beta_1=h_params[hp_config.HP_BETA_ONE],
    beta_2=h_params[hp_config.HP_BETA_TWO],
    epsilon=h_params[hp_config.HP_EPSILON],
)
model.compile(
loss=tf.keras.losses.CategoricalCrossentropy(label_smoothing=h_params[hp_config.HP_LABEL_SMOOTHING]),
    optimizer=opt,
    metrics=["accuracy", f1_score]
)
try:
    model.fit(
        train_data,
        epochs=h_params[hp_config.HP_EPOCHS],
        validation_data=valid_data,
        callbacks=callbacks,
    )
    model.save(model_path)
    ys_true, ys_pred, info = generate_results(
```

```
            shared_config.DATA_FOLDER,
            "",
            "encoder_spelling_bass_cut",
            chunk_size=h_params[hp_config.HP_CHUNK_SIZE],
            dataset="valid",
            verbose=True,
            model=model,
        )
        acc = analyse_results(ys_true, ys_pred)
        print(f"SAVED ACCURACY IS {acc}")
        return acc
    except ResourceExhaustedError:
        print("Model too big!")
        del model  # free up memory
        tf.keras.backend.clear_session()
        gc.collect()
        time.sleep(10)
        gc.collect()
        return np.nan


if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    gpus = tf.config.experimental.list_physical_devices("GPU")
    for gpu in gpus:
        tf.config.experimental.set_memory_growth(gpu, True)

    parser.add_argument(
        "-b",
        "--batch-size",
        default=32,
        type=int,
        help="Batch size for training"
    )
    parser.add_argument(
        "-e",
        "--epochs",
        default=250,
        type=int,
        help="Number of training epochs"
    )
    parser.add_argument(
        "-lr",
        "--learning-rate",
        default=0.001,
        type=float,
        help="Max learning rate used during training",
    )
    parser.add_argument(
        "-o",
        "--optimizer",
        default="adam",
        help="Type of optimizer to use during training",
        choices=["adam", "sgd"],
    )
    parser.add_argument(
        "-d",
        "--dropout-rate",
        default=0.33,
        type=float,
        help="Dropout rate used on Linear layers of neural network.",
    )
    parser.add_argument(
        "-dff1",
        "--dimension-ff-1",
        default=512,
        type=int,
        help="Dimension of Dense layer in Encoder Layers",
    )
    parser.add_argument(
        "-dm",
```

```
    "--d-model",
    default=70,
    type=int,
    help="Num of expected features in the encoder inputs",
)
parser.add_argument(
    "-dff2",
    "--dimension-ff-2",
    default=128,
    type=int,
    help="Dimension of Dense layer in Multi Task Networks",
)
parser.add_argument(
    "-mpe",
    "--max-pos-encoding",
    default=640,
    type=int,
    help="Maximum positional encoding"
)
parser.add_argument(
    "--model",
    default=0,
    dest="model_idx",
    action="store",
    type=int,
    help=f'index to select the model, between 0 and {len(shared_config.MODELS)}, '
         f'{[f"{n}: {m}" for n, m in enumerate(shared_config.MODELS)]}',
)
parser.add_argument(
    "--input",
    default=4,
    dest="input_idx",
    action="store",
    type=int,
    help=f'index to select input type, between 0 and {len(shared_config.INPUT_TYPES)},'
         f' {[f"{n}: {m}" for n, m in enumerate(shared_config.INPUT_TYPES)]}',
)
parser.add_argument(
    "-pt",
    "--pool-type",
    default='avg',
    type=str,
    help="Type of downsampling method.",
)
parser.add_argument(
    "-kc",
    "--key-chain",
    default=1,
    type=int,
    help="Flag indicating if model should use key predictions to make other predictions.",
)
parser.add_argument(
    "-r",
    "--relative",
    default=0,
    type=int,
    help="Flag indicating if model should use relative positional encodings.",
)
parser.add_argument(
    "-wu",
    "--warm-up-steps",
    default=0,
    type=int,
    help="Number of warm up steps in learning rate scheduler.",
)
parser.add_argument(
    "-md",
    "--max-distance",
    default=0,
    type=int,
    help="Maximum distance used when calculating relative positional encoding.",
```

```
)
parser.add_argument(
    "-dmm",
    "--d-model-multiplier",
    default=1,
    type=int,
    help="Factor to mutiply the model dimension by when calculating learning rate schedule.",
)
parser.add_argument(
    "-ls",
    "--learning-rate-scheduler",
    default='exponential',
    type=str,
    choices=['custom', 'exponential'],
    help="Determines which learning rate scheduler to use during training.",
)
parser.add_argument(
    "-cs",
    "--chunk-size",
    default=160,
    type=int,
    choices=[40, 80, 160, 320, 640, 1280],
    help="Dimension of each chunk when cutting sonatas in chord time-steps.",
)
parser.add_argument(
    "-dg1-nel",
    "--degree-1-nel",
    default=2,
    type=int,
    help="Number of encoder layers for the degree 1 encoder.",
)
parser.add_argument(
    "-dg2-nel",
    "--degree-2-nel",
    default=2,
    type=int,
    help="Number of encoder layers for the degree 2 encoder.",
)
parser.add_argument(
    "-q-nel",
    "--quality-nel",
    default=2,
    type=int,
    help="Number of encoder layers for the quality encoder.",
)
parser.add_argument(
    "-inv-nel",
    "--inversion-nel",
    default=2,
    type=int,
    help="Number of encoder layers for the inversion encoder.",
)
parser.add_argument(
    "-k-nel",
    "--key-nel",
    default=5,
    type=int,
    help="Number of encoder layers for the key encoder.",
)
parser.add_argument(
    "-r-nel",
    "--root-nel",
    default=3,
    type=int,
    help="Number of encoder layers for the root encoder.",
)
parser.add_argument(
    "-dg1-nah",
    "--degree-1-nah",
    default=3,
    type=int,
```

```
    help="Number of attention heads for the degree 1 encoder layers.",
)
parser.add_argument(
    "-dg2-nah",
    "--degree-2-nah",
    default=3,
    type=int,
    help="Number of attention heads for the degree 2 encoder layers.",
)
parser.add_argument(
    "-q-nah",
    "--quality-nah",
    default=4,
    type=int,
    help="Number of attention heads for the quality encoder layers.",
)
parser.add_argument(
    "-inv-nah",
    "--inversion-nah",
    default=1,
    type=int,
    help="Number of attention heads for the inversion encoder layers.",
)
parser.add_argument(
    "-k-nah",
    "--key-nah",
    default=1,
    type=int,
    help="Number of attention heads for the key encoder layers.",
)
parser.add_argument(
    "-r-nah",
    "--root-nah",
    default=1,
    type=int,
    help="Number of attention heads for the root encoder layers.",
)
parser.add_argument(
    "-dg1-rel",
    "--degree-1-relative",
    default=0,
    type=int,
    help="Indicates if relative positional encodings are to be used for the degree 1 encoder.",
)
parser.add_argument(
    "-dg2-rel",
    "--degree-2-relative",
    default=0,
    type=int,
    help="Indicates if relative positional encodings are to be used for the degree 2 encoder.",
)
parser.add_argument(
    "-q-rel",
    "--quality-relative",
    default=0,
    type=int,
    help="Indicates if relative positional encodings are to be used for the quality encoder."
)
parser.add_argument(
    "-inv-rel",
    "--inversion-relative",
    default=0,
    type=int,
    help="Indicates if relative positional encodings are to be used for the inversion encoder."
)
parser.add_argument(
    "-k-rel",
    "--key-relative",
    default=0,
    type=int,
    help="Indicates if relative positional encodings are to be used for the key encoder."
```

```
    )
    parser.add_argument(
        "-r-rel",
        "--root-relative",
        default=0,
        type=int,
        help="Indicates if relative positional encodings are to be used for the key encoder."
    )
    parser.add_argument(
        "-qc",
        "--quality-chain",
        default=1,
        type=int,
        help="Indicates if quality predictions are fed to the degree encoders."
    )
    parser.add_argument(
        "-b1",
        "--beta-1",
        default=0.95,
        type=float,
        help="Beta 1 parameter for Adam optimizer."
    )
    parser.add_argument(
        "-b2",
        "--beta-2",
        default=0.95,
        type=float,
        help="Beta 1 parameter for Adam optimizer."
    )
    parser.add_argument(
        "-eps",
        "--epsilon",
        default=1e-9,
        type=float,
        help="epsilon parameter for Adam optimizer."
    )
    parser.add_argument(
        "-lse",
        "--label-smoothing-eps",
        default=0.5275,
        type=float,
        help="epsilon parameter loss function label smooothing."
    )
    parser.add_argument(
        "-kd",
        "--key-dropout",
        default=0.15,
        type=float,
        help="Dropout to use in the key encoder stack."
    )
    main(args)
    args = parser.parse_args()
```

9.4 Performance and Visualisation Code

```
# PERFORMANCE
def plot_confusion_matrices(task_pred_dicts) -> None:
    """
    Given a dictionary full of prediction metrics, iterate through each task
    and create confusion matrices for them. Saves the figure to the working directory
    in a .png format.
    Args:
        task_pred_dicts (dict): Dictionary containing predictions and actual values.

    Returns:
        None
    """
    idx_to_labels = get_pred_label_opts(
```

```
        pitch_spelling=True, return_dict=True
    )
    cmap = sns.cubehelix_palette(
        start=0.5, rot=-0.75, as_cmap=True
    )
    for task_idx, task in enumerate(
        task_pred_dicts.keys()
    ):  # for each task
        labels = list(idx_to_labels[task_idx].values())
        task_cf_mat = confusion_matrix(
            task_pred_dicts[task]["actual"],
            task_pred_dicts[task]["predictions"],
            labels=labels,
        )
        fig, ax = plt.subplots(figsize=(5, 5), dpi=150)
        fmt = lambda x, pos: "{:,.0f}".format(x)
        sns.heatmap(
            task_cf_mat,
            annot=False,
            ax=ax,
            yticklabels=labels,
            xticklabels=labels,
            cmap=cmap,
            lw=0.5,
            cbar_kws={"format": FuncFormatter(fmt)},
        )  # add labels to heatmap axes
        ax.set_title(
            f"Confusion Matrix for {task.title()} Predictions",
            fontdict={
                "fontsize": 14,
                "fontweight": "medium",
            },
        )
        fig.tight_layout()
        plt.savefig(f"{task}_confusion_matrix.png", dpi=200)
        plt.show()


def get_f1_prec_rec(task_pred_dicts) -> dict:
    """
    Calculate weighted f1, recall, and precision scores for each task.
    A dictionary of dictionaries is constructed, and the metrics are added in a
    {'task': {'f1': 0.00, 'recall': 0.00, 'precision': 0.00}} format.
    Args:
        task_pred_dicts (dict):

    Returns:
        Dict of dict containing performance metrics.
    """
    task_perf_metrics = {k: dict() for k in FEATURES}
    for task_idx, task in enumerate(task_pred_dicts.keys()):
        task_actual = task_pred_dicts[task]["actual"]
        task_preds = task_pred_dicts[task]["predictions"]

        task_f1 = f1_score(
            task_actual,
            task_preds,
            labels=np.unique(task_preds),
            average="weighted",
        )
        task_recall = recall_score(
            task_actual,
            task_preds,
            labels=np.unique(task_preds),
            average="weighted",
        )
        task_precision = precision_score(
            task_actual,
            task_preds,
            labels=np.unique(task_preds),
            average="weighted",
```

```
        )

        task_perf_metrics[task]["f1"] = task_f1
        task_perf_metrics[task]["recall"] = task_recall
        task_perf_metrics[task][
            "precision"
        ] = task_precision

    return task_perf_metrics


def get_pred_label_opts(pitch_spelling, return_dict=False):
    """
    Get the tick_labels for each task, used for plotting. tick_labels
    array was constructed by Micchi et al in their code base and
    has been used here.

    Args:
        pitch_spelling (bool): Determines number of possible keys depending on pitch encoding method.
        return_dict (bool): Indicates if dictionary is constructed to make it easier to access.

    Returns:
        List or list of OrderedDict depending on return_dict parameter.
    """
    tick_labels = [
        KEYS_SPELLING if pitch_spelling else KEYS_PITCH,  # keys
        [str(x + 1) for x in range(7)] + [str(x + 1) + 'b' for x in range(7)]
        + [str(x + 1) + '#' for x in range(7)],  # DEGREE 1
        [str(x + 1) for x in range(7)] + [str(x + 1) + 'b' for x in range(7)]
        + [str(x + 1) + '#' for x in range(7)],  # DEGREE 2
        QUALITY,  # QUALITY
        [str(x) for x in range(4)],  # INVERSION
        PITCH_FIFTHS if pitch_spelling else NOTES,  #
        PITCH_FIFTHS if pitch_spelling else NOTES,  # ROOT
    ]

    if return_dict:  # optionally return an ordered dict of each task as opposed to an array
        tick_labels = [
            OrderedDict(
                {k: v for k, v in enumerate(tick_labels[index])})
            for index in range(0, 6)
        ]
    return tick_labels


def get_pred_labels(y_true, y_pred):
    label_dicts = get_pred_label_opts(pitch_spelling=True, return_dict=True)
    output = dict()  # create empty output array for each task's prediction labels
    for j in range(len(y_true)):  # for each task

        task = FEATURES[j]
        tmp_preds = np.argmax(y_pred[j], axis=-1)  # get the logits for the preds
        tmp_true = np.argmax(y_true[j], axis=-1)  # same for true labels

        idx_to_label = label_dicts[j]  # get dictionary that translates to label from index
        tmp_pred_labels = np.vectorize(idx_to_label.get)(tmp_preds)  # map the dictionary to both
        tmp_true_labels = np.vectorize(idx_to_label.get)(tmp_true)

        tmp_output = {'predictions': tmp_pred_labels, 'actual': tmp_true_labels}

        output[task] = tmp_output
    return output
```