­

Jazz Piano Lead Sheet Arrangement via Deep Learning

Elliot Hogg

200817750

David Herbert

Mark Turner

Jim Mcgrath

DD/mm/yyyy

15000 Words

Declaration

I hereby declare that this dissertation represents my own work except where otherwise stated.

Abstract

Over the past 10 years, deep learning has been extensively researched and used in the task of music generation. However, one area that remains mostly untouched, is the task of lead sheet arrangement. In the genre of Jazz, most music is represented in the lead sheet format – a single notated melody and chord symbols. The interpretation and arrangement of the music must be decided and undertaken by the performer. To tackle this problem, a novel conditional generative adversarial network (C-GAN) was designed to generate jazz chords which could be used to create full arrangements. Participant lead evaluations showed that resulting arrangements were found to be more effective than template chord selection.

Acknowledgments

I would firstly like to thank my supervisor David Hebert, whose support and advice was invaluable to this project.

I would also like to thank Mark Turner for his continued help and support throughout the project.

Contents

1. Introduction (1700 words)

2. Background Research (1700 words)

(6600 words):

4. Implementation

4.1. XML Chord Scraper

4.2. Deep Learning Models

4.3. Lead Sheet Arranger

4.4. Complete System ??

(3300 words):

5. Results and Evaluation

5.1 Scraper Accuracy

5.2. Deep Learning

5.3. Arrangements

5.4. complete system

5.3.1. Participant survey

5.3.2. Technical evaluation

(1700 words):

6. Conclusion

6.1. Meeting aims and objectives

6.2. Effectiveness of approach and systems

6.3. Contribution to the field

6.4. Future Considerations

1. Introduction

Lead sheet arrangement is a task whereby a musician takes an abbreviated representation of a song and adds harmonic and rhymical elements to transform it into a compelling piece of music. A lead sheet consists of two elements - the songs melody, and chord symbols. The melody is notated using western music notation (glossary) and is played by the right hand of a pianist. The chords symbols are notated as letters positioned above the melody and represent an abbreviated notation of what the left hand will play. Figure 1 shows the first 4 bars of a piano lead sheet.

A black and white photo of a piano

Description automatically generated with low confidence

Figure 1. The first 4 bars of the lead sheet for the song Over the Rainbow by Errol Garner. The melody is represented as western notation, and the chord types are indicated by the chord symbols above.

Lead sheet arrangement can be divided into two subtasks: (1) Harmonic – choosing a set of notes to be played for each chord symbol. This task is also referred to as *chord voicing*. (2) Rhythmic – temporally arranging the chosen chord notes to form a rhymical pattern. Figure 2 highlights these two subtasks. The notes on the lower set of lines now instruct the left hand on what to play, and together with the right hand melody form a full arrangement.

Diagram, engineering drawing

Description automatically generated

Figure 2. Lead sheet arrangement of the first bar from figure 1. From left to right, a C chord symbol (shown in red) is first represented harmonically, and then arranged rhythmically.

In the area of jazz music, the lead sheet is the traditional and most common way for songs and pieces of music to be notated. This means that for a pianist to learn a jazz song, they must either find an arranged version, or arrange the lead sheet themselves. However, due to the creative and interpretive nature of jazz music, it is often encouraged that jazz pianist should arrange their own lead sheets.

Computationally, lead sheet arrangement can be defined as a process that takes a lead sheet as input, and outputs a harmonically and rhythmically complete digital representation of that song.

Jazz musical theory defines a standardised set of possible notes that can be included in chord voicings(citation). When choosing the notes that will form a chord symbols voicing, the musician is restricted to that standard set of notes. However, as there are so many notes on the piano, the order of these notes can vary enormously. Furthermore, not all of the standardised set of notes that make up a chord are always included.

There are a few commercial applications, such as *Band in a box* and *iRealPro,* that perform the harmonic element of lead sheet arrangement, however they are reliant on simple selection algorithms and a limited dictionary of chord voicings. The chord voicings themselves are generic and lack musicality. There are however no applications that attempt to perform the rhythmic subtask of lead sheet arrangement, arguably due to this being a much more complex task (citation).

There has been some academic research of traditional programming approaches to the harmonic task of lead sheet arrangement that have shown to outperform the aforementioned applications, however the results of traditional programming approaches lack variation, as they struggle to output multiple different sets of notes for the same chord symbol (citation). They also rely on being preprogramed with a set of musical theory rules.

The use of deep learning in lead sheet arrangement has seen very little academic research. There exists only two publications on the topic, both of which have the same authors. There has also been some recent publications using deep learning in the task of chord voicing generation, which is essentially the same task as the harmonic arrangement task in lead sheet generation. These papers are critically evaluated in section 2.x.

Although there is limited research on the task of music arrangement, the task of musical generation has seen a large amount of attention (citation). This literature is valuable to this project, as arrangement can be defined as a subtask of generation. Surveying an evaluating recent music generation research can greatly contribute to the direction of this project. Especially in regard to gaining an insight into which deep learning models would be most effective for this project. Section 2.x surveys music generation literature in search of effective deep learning algorithms.

And the focus on music composition within the field of deep learning has

has regained a large amount of academic and public attention within the past 10 years (citation). This is primarily due to the development of deep neural networks (Ji et al., 2020: 5). One great example of such generation is *DeepBach*, a deep learning model that can compose novel, convincing chorales in the style of composer Bach (Hadjeres et al., 2017). Deep learning has also been used to generate monophonic and polyphonic music of many different genres (citations needed).

Research and implementation of deep learning in the task of lead sheet arrangement is however very limited. There exist only two publications on the topic, both from the same authors. In their research, they employed deep learning to arrange pop lead sheets into a full arrangement for a 4 piece band (reference).

Music generation tasks are usually achieved using either recurrent neural networks (RNN) or generative adversarial networks (GAN). The former type of network is highly effective at generating temporal sequences, as the networks outputs are dependent on previous outputs (citation). The latter network is effective at generating completely new instances of data that are undistinguishable from a given set of data (citation).

The inherent nature of the lead sheet is that, once a musician has composed an arrangement, they tend to memorise it rather than notate it (citation). This is a particular issue in jazz, in which there exist very few arrangements of jazz standards in relation to the size of the total corpus (citation). For intermediate or amateur jazz musicians, or for advanced classically trained musicians, the task of lead sheet arrangement can be challenging, and can create an effective barrier to entry for the genre as a whole (reference).

Harnessing the power of deep learning in order to create a system capable of generating compelling arrangements would be invaluable to the jazz community, as it would make jazz music much more accessible.

The focus of this project is to create a lead sheet arrangement system that will take a lead sheet as input and output an arrangement. The system will also be intended to provide a framework for others to employ and use in order to advance the field of research. The system will be limited in that it will only address the harmonic element of the process, i.e. It will provide chord voicings for the chord symbols, and not arrange those voicings temporally.

The deep learning model will require a large dataset of jazz chord voicings and their associated chord symbols. An extensive search of existing datasets presented no such dataset, therefore this research paper will also aim to gather and present a novel “jazz-chords dataset”.

In order to gather this dataset, a chord scraping tool will need to be used on existing jazz piano solo arrangements. A tool will also be required to combine generated chords and lead sheets together to form full arrangements as output. Both of these tools will also be developed as part of this project, as no existing systems that perform these tasks currently exist.

The system will consist of three key elements. (1) A chord scraper that is capable of extracting chord voicings from fully arranged songs with chord symbols. This scraper will be limited to songs in MusicXML format, which is a popular XML based music encoding language (reference). (2) A chord generator that will use a conditional generative adversarial network to generate musically pleasing and convincing chord voicings. The network will be trained using a novel dataset of labelled chords gathered by using the chord scraper on existing jazz piano arrangements. (3) A lead sheet arranger, which will take the melody from the lead sheet and the chords from the generator, combine them, and output an arrangement in MusicXML, MIDI, and Audio formats.

The project will also present the novel dataset of labelled chord voicings for use in further deep learning tasks.

**1.2 Aim**

To create a deep learning led system that can generate arrangements of any given jazz lead sheet.

**1.3 Objectives**

* To develop a program that can scrape chord voicings from fully arranged pieces of piano music with chord symbols in MusicXML format
* To present a novel dataset of labelled jazz piano chord voicings
* To create a conditional generative adversarial network that can output compelling colourings and voicings for chord symbols from a given lead sheet
* To develop a program that can combine chord voicings with the melody of a lead sheet in a way that is both convincing and playable.
* To perform some empirical and participant led testing on the outcomes.

**1.4 Outline**

The structure of the report follows both the order it was developed in as well as the order in which the system as a whole operates.

* **Chapter 2** provides a review of the recent literature within the field of deep learning as it pertains to music. There will also be a critical evaluation of existing lead sheet arrangement models.
* **Chapter 3** provides information about the data used in the project.
* **Chapter 4** provides a deep dive into how the three key components of the system were conceptualised and developed. Starting with the chord scraper, then the C-GAN deep learning model, and finally the lead sheet arranger.
* **Chapter 5** presents an evaluation of each component, as well as an evaluation of the results of the system as a whole. Evaluations of the system are both technical and participant led.
* **Chapter 6** presents a conclusion of the project, with commentary on the how initial aims and objectives were met, as well as some future considerations and recommendations for further research.

2. Background Research

2.1 Motivations

The initial motivations for this project were to make a meaningful and novel contribution to the jazz piano community within the context of a computer science project. As a keen jazz pianist with a classically trained background, the task of lead sheet arrangement has always been a challenge. In order to make an arrangement sound good, a large amount of knowledge and experience in voicing chords is required. A system that could generate chord voicings that were indistinguishable from the voicings of renowned pianists such as Bud Powell, Bill Evans, or Herbie Hancock would remove the barrier for entry to a large number of aspiring jazz pianists.

This is the motivation that led me to pursuing a research project in the field of deep learning and lead sheet arrangement. I hope that this research can at the very least, serve as a springboard for future research to be carried out in this area.

2.2 Searching for chord scraping and lead sheet arrangement tools

As detailed in section 1, the deep learning model in this project will require a large dataset of pairs of chord voicings and associated chord symbols. As no such dataset exists, it must be gathered from existing fully arranged jazz piano solos. A search of the literature provided no instances of any previous research that had developed a scraper that was capable of such a task. Further searches of existing software and GitHub repositories presented 3 tools which were capable of extracting chord symbols from sheet music in varying formats, however the tools did not extract any chord voicings (reference). The lack of a chord scraping tool could be explained by the following reasoning. As in this project, the motivations for developing such a tool would be to extract a dataset for use in a musical chord related deep learning task. Furthermore, the majority of the deep learning in music field is concerned with researching music generation. The deep learning models used in this task generally require examples of previous music and are not concerned with the relationship between chord voicings and symbols.

This project also requires a tool that can take chord symbols from a lead sheet, pass them into a deep learning model, take the resulting generated chords and finally combine them with the melody of the lead sheet to form a full arrangement. The requirements and functionality of this tool are very specific to the requirements of this research project, and thus finding an existing system that could be used or adapted for this task was unlikely. A thorough search of the literature, available software, and GitHub confirmed this assumption.

2.3 Searching for an effective deep learning model

As mentioned in the introduction, there is a limited amount of previous research on the task of lead sheet arrangement using deep learning. However, as arrangement is a subtask of generation, research on music generation is also highly relevant to this project. In this subsection, lead sheet arrangement research and music generation literature will be surveyed in order to assess the capabilities of current research and see how effective their models are in the task of arrangement. In doing this research, the aim is to both find pre-existing models that can be adapted to fit the needs of this project, as well as use the designs and results of existing models to guide the design within this project of a partially novel deep learning that can accomplish the task of lead sheet arrangement.

In order to thoroughly survey the landscape of arrangement and generation, a full scrape of the field was conducted, which were then filtered and reduced into a list of 35 research papers. Meta information was manually extracted from each paper, such as the deep learning models used, datasets used, and whether the models were publicly available. This table can be found at APPENDIX X.

The first paper of note was titled ‘Lead Sheet Generation and Arrangement by Conditional Generative Adversarial Network’. In this project, the authors were able to generate lead sheets, and then arrange them both harmonically and rhythmically. As the title suggests, the model used was a conditional generative adversarial network or C-GAN. The results of the paper demonstrated that this model is effective in the task of harmonic arrangement, as it was able to generate convincing chord voicings when given chord symbols.

One major limitation of this model however was that it could only generate simple chords, i.e., major or minor. It was not capable of more complex jazz chords, due to the fact that it was trained on pop songs.

Despite the limitations of this research, the results were promising. The model was available on GitHub, however, attempts to get it running failed as the list of dependencies was incomplete, and some versions of those dependencies were unavailable. Contact was attempted with the authors; however, no response was received.

This paper was very useful to this project, as it brought to our attention a deep learning model that was capable of the task of lead sheet arrangement.

A further 3 papers were evaluated in which the authors used either generative adversarial networks (GAN’s) or C-GAN’s in the task of music generation (Dong et al., 2017; Yang et al., 2017, Liu et al., 2018). Each model was able to generate novel instances of music that were indistinguishable from the training dataset.

The second paper of note was titled ‘Chord Jazzification: Learning Jazz Interpretations of Chord Symbols’. Although this paper made no mention of the lead sheet, the model presented was able to take as input a sequence of chord symbols and output a series of chord voicings to represent those symbols, which is fundamentally the same task as harmonic lead sheet arrangement.

The main insight gained from this paper was the way in which the chord symbols and chord notes were encoded. The authors presented an effective way on encoding chord symbols as numerical labels. This approach provided a useful framework for which this research project could both gather and represent its own data, of which the details are presented in section 3, 4.1 and 4.2.

The model used to generate the chords sequences were variations of recurrent neural networks (RNN’s). The reason this network was used is that it excels at generating sequential data in which each individual data instance is dependent on the instance generated before it. This type of model would be highly effective in this project aim of harmonic lead sheet arrangement, as it both generates individual chord voicings as well as ensures that they are in some way meaningfully connected as a sequence.

The one major limitation of the network was that the chord voicing sequences it generated were limited to a maximum of 8 bars in length, which is far less than a jazz piano lead sheet. The source code for this model was available on GitHub, and was downloaded and in order to investigate further.

After attempting to increase the length of the generated sequences, it was discovered that the maximum length of generated sequences was limited to the length of the sequences used to train the model. Due to a lack of data, the authors were forced to split their data into training instances of 8 bar long sequences.

This discovery suggested that RNN’s would not be suitable to the requirements of this project, as the sheer amount of data required to train them in order to output full length lead sheet arrangements would be unfeasible.

A further 7 research papers using RNN’s in the task of music generation were evaluated, confirming that the length of sequence generation was limited to the length of training sequences (Hadjeres et al., 2017; Liang et al., 2019; Mogren, 2016; Szelogowski 2021; Teng et al., 2017; Zhao et al., 2020; Zhu et al., 2020;). All of these papers can be found at APPENDIX.X.

Two further generative deep learning algorithms were highlighted in a journal titled “A Comprehensive Survey on Deep Music Generation: Multi-level Representations, Algorithms, Evaluations, and Future Directions” (Ji et al., 2020). These two algorithms were Autoencoders (AE’s) and Variational Autoencoders (VAE’s). Although these algorithms showed some good results in the task of music generation, the results of three recent papers suggest that VAE’s are not as effective as GAN’s, mainly due to the fact that they do not appear to be able to learn the true posterior distribution of a given dataset (Angioloni et al., 2020; Engel et al., 2017; Valenti et al., 2020).

In order to learn more about C-GAN’s and their use in generation tasks, a wider scope of literature was searched. There were two papers of note, which both used C-GAN’s to perform tasks that were fundamentally the same as harmonic lead sheet generation.

The first paper presented the *Pix2Pix: image translation model,* which uses a C-GAN to perform a variety of image translation tasks. For example, in one instance, the model is trained to take as input a satellite image and output a google maps style image representation. Although the data type and shape differ to that of chord symbols and notes, the translation from one representation to another applies to this project, and thus the model is highly informative and applicable.

The second paper presented a one-dimensional adaptation of the *Pix2Pix* model. This is also of significant interest to this research project, as both chord labels and chord notes are best represented as one dimensional vectors. Although this adapted model was not openly available, the author was contacted and enthusiastically agreed to share

Both of the above models are used within this research project, as seen in section 4.2.

2.4 Generative Adversarial Networks (GAN’s)

Generative adversarial networks are a type of neural network developed by Ian Goodfellow and his colleagues in 2014 (Goodfellow et al., 2014). For the reader to understand this paper, a high-level explanation of GAN’s will be provided. The purpose of a GAN is to be able to generative new instances of data from a given data distribution. For example, if a GAN was trained on a dataset of images of faces, it would then be able to generative completely new images that would exist in a similar distribution space to the training dataset.

A GAN is made up of 2 independently functioning neural networks, a discriminator, and a generator. (Goodfellow et al., 2014: 3).

The generator is a neural network that takes a noise distribution as input, and outputs a fake instance of data. To use the previous example, the generator would output fake images of faces. Initially, the fake images would be very poor, however through training the generator, it would eventually be able to output fake images that were convincing enough to be indistinguishable from images in the dataset.

The discriminator is a neural network that operates as a classification network. It takes as input both real instances of data, and generated, fake instances of data, and outputs a number indicating whether the input was real or fake.

When initially training the GAN model, the discriminator easily classifies the real instances as real, and the generated instances as fake. However, throughout training, the generator improves to produce more and more convincing instances, to the point where the discriminator is unable to effectively classify real instances as real, and fake instances as fake. At this point, the generator can produce completely novel instances of data from the given dataset distribution.

2.5 Conditional Generative Adversarial Networks (C-GAN’s)

Conditional generative adversarial networks allow the output of GAN’s to be conditioned on a given input (citation). Extending the example from 2.2 to having a labelled dataset of faces made up of faces of men, woman, girls, and boys. When training the generator, instead of passing in a noise distribution, the labels are passed in. In doing so, the generator is conditioned to effectively learn the distributions of each of the four labels. Thus, when generating faces using the trained network, one of the four labels could be passed into the generator, resulting in a generated face of that labels distribution.

2.6 Survey of available datasets

Deep learning generation models require a large amount of data for training. C-GAN’s require data that is labelled. For this research project, the data must consist of pairs of chord symbols with their respective chord voicings, i.e. the set of notes that represent a chord symbol. For more details on the data requirements, see section 3 and 4.1.

In other deep learning fields, such as image generation, extended research over the last decade has left a great number of refined corpuses for which researchers can use in their models. However, as the field of deep learning in music is still relatively small, the availability of datasets is not comparable. In order to determine if there was a pre-existing dataset that could be used or adapted to fit the model of this project, a full survey of available datasets was conducted. The survey found 15 jazz related datasets. However, after conducting a survey, a much more comprehensive list of datasets was found in Ji et al’s., 2020 deep learning in music review paper (Ji et al., 2020:56-58). Therefore, only the datasets that are not present in Ji et al.’s paper will be presented below:

|  |  |  |
| --- | --- | --- |
| Format | Type | Contains |
| Text | Jazz Audio-Aligned Harmony (JAAH) Dataset | Meter, structure, and chords of 113 Jazz tracks |
|  | Weimar Jazz Database (WJazzD) | Transcriptions of 135 jazz solos |
|  | JazzCorpus | Annotated chord progressions for 77 jazz pieces |
|  | Chord-Jazzification dataset | Annotated chord voicings from 50 pop-jazz piano solos |
| MIDI | LMD-matched | 45,129 songs matched to the Million Song Dataset |
|  | Big\_Data\_Set: “The Largest MIDI Collection on the Internet” | 130,000 songs including jazz solos |
| MusicXML | Charlie Parker's Omnibook data | Transcriptions of 50 Charlie Parker improvisations |

Table 1. A list of jazz related datasets that are available and open to use. This is an extension of Ji et al.’s 2020 dataset survey.

Following a survey of all the available datasets, 2 were identified as being of interest to this paper. (1) The Wikifonia dataset, which is a collection of 7000 jazz lead sheets. This dataset can be used in order to test the project. (2) The Chord-Jazzification dataset (see table 1). This is the only existing dataset that contains chords presented as pairs of chord symbols and chord voicings. The only limitation is that most of the chords are taken from pop chords, and do not contain the more complex chords used in jazz.

Following this survey, it was determined that a novel dataset would need to be gathered in order to train the deep learning model. More information of this can be found in section 3 and 4.1.

2.7 Technologies & Design Patterns

3. Gathering a Library of jazz piano solo arrangements

3.1 Initial Research

Training conditional adversarial networks (C-GAN’s) require a large amount of labelled data (see 2.3 and 2.4). In the context of this project, the data is chord voicings, and the labels are their associated chord symbols. Figure 3 shows how these data pairs would be represented inside a jazz piano fully arranged score (glossary).

Diagram

Description automatically generated with medium confidence

Figure 3. Showing two data pairs, with chord symbols highlighted in red, and chord voicings highlighted in blue. The melody notes have been omitted from this score.

A review of available datasets (section 2.5) found one corpus which contained chord symbols and chord voicings; however, the chords were taken from pop songs and thus didn’t meet the requirements for this project (reference to pop vs jazz chords?).

In order to gather a novel dataset of labelled chord voicings, a large library of jazz piano solos would need to be scraped and the chords extracted. The solos would also need to be of a certain degree of quality, in that they would need to contain chord symbols that were accurate and chord voicings that were compelling.

Music is conventionally represented in either MIDI or an XML type format such as MusicXML. As MIDI does not allow for chord symbol encoding (reference), a library of jazz piano solos in XML format was required.

An extensive survey was conducted in which 23 XML libraries were discovered (table from <https://www.musicxml.com/music-in-musicxml/> in Appendix). The survey showed that a website called *MuseScore.com* had over 1 million scores in XML format, significantly more than any of the other libraries.

The search function on *Musescore* showed a total of 4,837 available jazz piano solos. However only 300 of the solos were fully arranged and contained chord symbols. The rest were either lead sheets, or full arrangements without chord symbols. Of the 300 suitable solos, 171 were selected. Further details of this are found in the following section.

3.2 Fully Arranged Jazz Piano Solo Library

171 fully arranged jazz piano solos with chord symbols were manually downloaded from *musescore.com*. In order for a solo to be selected, it must have met all of the following criteria:

1. Fully arranged, i.e., both the treble and bass clef present

2. Have chord symbols

3. Composed by a recognised jazz musician

4. Arranged by a reputable user with a peer reviewed score of 4.7 stars out of 5 or above

\*\* Maybe insert table showing composers?\*\*

The solos are in MusicXML format, which is an XML-based format. A detailed list on the dataset can be found at Appendix.X.

3.3 MusicXML

In order to understand section 4, a brief explanation of MusicXML is provided in this section. For a more detailed understanding, see Good (2001).

MusicXML is an XML based markup language for representing musical scores. Like MIDI, MusicXML is standardised, and can be interpreted by all major music notation software and transformed into a score.

Figure 4 below shows how the first chord symbol and voicing pair in figure 3 is represented in MusicXML format.

Looking at figure 4, the chord symbol is represented by the use of a harmony element. The inner elements indicate the root of the chord and the kind of chord. The harmony element can contain additional elements for more complex chords.

The notes that make up the chord voicing are represented by the note elements that follow. The “default-x” attribute is helpful in determining which notes make up a chord voicing, as chord notes are always aligned vertically.

<harmony print-frame="no">

<root>

<root-step>C</root-step>

</root>

<kind text="M7">major-seventh</kind>

</harmony>

<note default-x="92.47" default-y="-40.00">

<pitch>

<step>E</step>

<octave>4</octave>

</pitch>

<duration>4</duration>

<voice>1</voice>

<type>whole</type>

<staff>1</staff>

</note>

<note default-x="92.47" default-y="-30.00">

...

</note>

<note default-x="92.47" default-y="-130.00">

...

</note>

<note default-x="92.47" default-y="-110.00">

...

</note>

<note default-x="92.47" default-y="-100.00">

...

</note>

Figure 4. A C Major 7th chord symbol and notated voicing represented in MusicXML format

4. Chord Scraper

This section will first detail the chord scraper system developed and used within this project. A high-level overview of the system will be provided as well as a detailed report of how the system was developed. Following this, the extracted jazz-chords dataset will be presented and evaluated. Finally, there will be an evaluation of the effectiveness of the Chord Scraper system. Sections 3.2 and 3.3 are recommended precursors to this section.

4.1 Requirements

- To extract pairs of chord symbols and the group of notes that make up the chord voicing from MusicXML solo piano scores.

- To output raw data as well as data that has been partially cleaned and refined (see section 4.1.2).

- To work on all solo piano scores in MusicXML format including previously unseen scores.

-To be able to scrape a large number of scores (>100) in a reasonable amount of time (< 5 minutes).

- To extract the group of notes that represent a chord symbol with a high degree of accuracy ( > 97%).

- To have a highly usable command line interface that enables other researchers to use the tool.

4.2 System Overview

As seen in figure 5, the chord scraper takes a number of piano scores in MusicXML format, and outputs chord data in 2 different formats. The first output format represents the chords exactly as they were inputted, whereas as the second transforms the chords into a format that it more suited to machine learning model inputs. More information of this can be found in section 5.1.

Diagram

Description automatically generated

Figure 5. A system diagram showing the chord scraper system

In order to highlight the workings of the system, a small scraping example is presented below.

Figures 6 and 7 show a 2-bar extract from a jazz piano solo as a score view and as MusicXML. Figures 8 and 9 demonstrate the result of passing the MusicXML from figure 6 into the chord scraper. In figure 8, the chord voicings are represented as both a list of note names as well as their associated note numbers (reference/glossary/footnote?). As seen in figure 8, the data is a true representation of its input. However, in figure 9, chords 1 and 3 have been transposed (glossary) to have a root of C. This then means there is no need to include root information or note labels.

Schematic

Description automatically generated with medium confidence

Figure 6. A 2 bar score representation of a MusicXML jazz piano solo score.

<measure number="3" width="280.43">

<harmony print-frame="no">

<root><root-step>G</root-step></root>

<kind text="m7">minor-seventh</kind>

</harmony>

<note default-x="29.66" default-y="-15.00">...</note>

<note default-x="29.30" default-y="-145.00">...</note>

<note default-x="29.30" default-y="-135.00">...</note>

<note default-x="29.30" default-y="-125.00">...</note>

<note default-x="29.30" default-y="-115.00">...</note>

...

<harmony print-frame="no">

<root><root-step>C</root-step></root>

<kind text="7">dominant</kind>

<degree>

<degree-value>5</degree-value>

<degree-alter>-1</degree-alter>

<degree-type>alter</degree-type>

</degree>

</harmony>

<note default-x="153.20" default-y="-20.00">...</note>

<note default-x="153.20" default-y="-145.00">...</note>

<note default-x="153.20" default-y="-135.00">...</note>

<note default-x="165.07" default-y="-130.00">...</note>

<note default-x="153.20" default-y="-120.00">...</note>

...

<harmony print-frame="no">

<root><root-step>F</root-step></root>

<kind text="7">dominant</kind>

</harmony>

<note default-x="246.05" default-y="-25.00">...</note>

<note default-x="246.05" default-y="-150.00">...</note>

<note default-x="246.05" default-y="-140.00">...</note>

<note default-x="246.05" default-y="-130.00">...</note>

<note default-x="246.05" default-y="-120.00">...</note>

</measure>

<measure number="4" implicit="yes" width="180.55">

...

<harmony print-frame="no">

<root><root-step>C</root-step></root>

<kind text="7">dominant</kind>

<degree>

<degree-value>5</degree-value>

<degree-alter>1</degree-alter>

<degree-type>alter</degree-type>

</degree>

</harmony>

<note default-x="71.49" default-y="-55.00"></note>

<note default-x="71.49" default-y="-40.00"></note>

<note default-x="71.49" default-y="-30.00"></note>

<note default-x="71.49" default-y="-165.00"></note>

<note default-x="71.49" default-y="-130.00"></note>

...

Figure 7. A 2 bar excerpt of a MusicXML jazz piano solo.

root, type , extensions , note\_numbers , notes

1 G , minor-seventh, "[]" , "[23,26,30,33,52]", "[G2,Bb2,D3,F3,C5]"

2 C , dominant , "[{'degree':'9','alter':'-1','type':'add'}]", "[22,26,28,32,50]", "[Gb2,Bb2,C3,E3,Bb4]"

3 F , dominant , "[]" , "[21,25,28,31,49]", "[F2,A2,C3,Eb3,A4]"

4 C , dominant , "[]" , "[16,28,38,44,48]", "[C2,C3,Bb3,E4,G#4]"

Figure 8. Raw chord data representing score in figure 6 (CSV format)

type , extensions , note\_numbers

1 minor-seventh, "[]" , "[16,19,27,26,45]"

2 dominant , "[{'degree':'9','alter':'-1','type':'add'}]" , "[22,26,28,32,50]"

3 dominant , "[]" , "[16,20,23,26,44]"

4 dominant , "[{'degree':'5','alter':'1','type':'alter'}]", "[16,28,38,44,48]"

Figure 9. Cleaned and transposed chord data representing chords in figure 6 (CSV format)

4.3 Development

The chord scraper system was developed using the Python programming language. The reason for selecting Python was both its effectiveness as a scripting language and its array of inbuilt data manipulation packages.

The initial aim when developing the system was to write a script that could effectively and consistently extracting chord information from MusicXML files. Following this, an additional script would be developed in order to transform the extracted data into a format that would be more suitable to the machine learning model used for chord generation, as well as for other potential users.

For the purposes of readability, these two scripts will be presented as separate entities, however it must be noted that in reality they are to some extent, interconnected, and therefore exist inside the same Python file. The chord scraper can be found inside the *chord\_scraper* directory at the root of the project.

4.3.1 Chord Extraction

The first functioning version of the script worked by iterating through each line in each file and using Regex to extract the required data. The script worked by first finding a harmony element and extracting the chord symbol information. The way in which MusicXML is formatted means that the note element after each harmony element is the top note of that chord. However, because MusicXML lists all of the treble clef (glossary) notes before the bass clef notes, finding the remaining notes of a chord voicing using the order of elements was not possible. Figure 10 highlights this issue.

Diagram, schematic

Description automatically generated

Figure 10. A two-bar score showing the line order of MusicXML elements

After further analysing the MusicXML markup syntax, it was discovered that each note element had a “default-x” attribute, which denoted the notes *x*-axis coordinate. By using the “default-x” attribute of the note directly after each harmony element, the other notes that made up that chord could be found. This is because notes that make up a chord voicing are usually vertically aligned. The issue with this approach is that there are cases in which some of the notes in a chord are offset on the *x* axis. This would mean that some notes were not included in chord voicings when they should have been.

The first cause of this is when there are 2 notes in a chord that are next to each other on the piano, and vertically aligning them would cause the notes to overlap, and thus be unreadable. Figure 11 highlights this issue.

Diagram, schematic

Description automatically generated

Figure 11. A score highlighting the notes selected to make up a chord voicing using “x-location” value.

The second cause of this issue is when chord voicings are temporally arranged to form a rhythmic pattern. This means that the notes that make up the chord voicing are not stacked on the x-axis, but adjacent to one another. This is highlighted in figure 12.

Diagram, schematic

Description automatically generated

Figure 12. A chord voicing in which the notes are arranged in a rhythmic pattern.

In order to solve the first issue, a concept of *x-axis tolerance* was introduced to the chord extraction script. This allowed for notes that were within *x* amount of the “x-location” to be identified as part of the chord voicing. For example, if the first note element after a harmony element had a “x-location” value of 100, and the *x-axis deviation* was set to 10, then any note with an “x-location” value of between 95 and 105 would be extracted as part of that chord voicing. An *x-axis deviation* value of 20 degrees ensured that all chord voicing notes were extracted, whilst ensuring that notes either side of the chord voicing were not. This is highlighted in Figure 12.

Diagram, schematic

Description automatically generated

Figure 12. The use of *x-axis deviation* to extracted all of the notes in a chord voicing.

It was not possible to use *x-axis deviation* to successfully extract of chord voicings arranged in a rhythmic pattern. For example, if the deviation was increased to be able to extract rhythmically arranged chord voicings, it would then extract unwanted notes for vertically stacked chord voicings. No other solution was found to extract rhythmically arranged chord voicings.

The issue with using this method to extract chord voicings, is that not all

Although this first approach functioned as expected, iterating through it line by line using Regex seemed like bad practice as XML is a structured markup language. After some further research, it was clear that a much more effective approach would be to parse each MusicXML file into a searchable data structure (citation?). Python’s build in XML API – *ElementTree* was identified as a suitable tool, as it was able to transform XML into an ordered Tree data structure. This would allow for search operations to be performed in constant time [O(1)] as oppose to linear time [O(n)] when iterating through *n* number of lines per file.

The second and final major version of the script used *ElementTree* to transform each MusicXML file into an ordered tree data structure. Rather than going through each file line by line, the second version traversed through the tree, bar by bar. Each bars children were then searched to extract each harmony element and their associated note elements to form pairs of symbols and voicings. The MusicXML tree structure can be seen in figure 12. The technique of finding the note immediately after a harmony element and using its “default-x” attribute to find all of the other notes was again used in this version. However, as each bars children were indexed, the implementation was much cleaner.

Diagram

Description automatically generated

Figure 12. MusicXML in as an Ordered Tree Data Structure

4.3.2 Chord Data Manipulation

After developing a functioning chord extraction script, some additional data manipulation functions were created in order to improve the output of the chord scraper. For example, as chords by definition must have 3 or more notes, the gathered chord voicings that had less than 3 notes were removed. Also, the notes of each chord voicing were additionally represented using note numbers, a system in which the numbers 1-88 represent each note on the piano (reference). Outputting chord voicings in this way would allow other users to easily encode the data for further use. The chord voicings were also subjected to a small degree of pitch normalisation, in which any very high or low notes were moved further towards the middle.

Finally, all of the chords were transposed so that their root note was C. For example, every note in a D major-seventh chord would be moved down 2 notes (figure? Or more explanation). This then meant that there was no distinction between root notes, and only between chord types. For example, rather than having 200 C major-sevenths, 150 E major-sevenths, 400 F major-sevenths, and 250 Ab major sevenths, the data would consist of 1000 major-sevenths. Therefore, when using the dataset to train a machine learning model, there would be much more examples of each chord type. There is also no downside to this approach, as any generated chords can be transposed back to the root note of the user’s choice.

The decision to not perform some of these data manipulation steps when encoding the data for the machine learning model were as follows. The use of this chord scraper in the future by other researchers or users would most likely be for the purposes of extracting data for use in machine learning or data analysis. Therefore, it seemed like a sensible decision to output chord data in a state that would be desirable for such users. As noted in section 4.2, the chord scraper also outputs the data in raw format, before any data transformations have taken place.

Initially, the two datasets were outputted as data pickles (glossary). This was due to the fact that pickles make extracting and reimporting Python lists and dictionaries extremely straight forward. However, as many other programming languages do not offer in build support for data pickles (reference), the datasets were additionally outputted in CSV format. It must be noted however, that storing Python data structures in this format is less than ideal.

The chord scraper also features a simple command line interface (CLI), which was implemented using the Python library *argparse.* Users can specify an input and output path, as well as enable error logging and the printing of chord meta information. Further details of this can be found here: (github link).

For an evaluation of the results and effectiveness of the chord scraper system, see section 5.1.

4.4. Results and Evaluation

In this sub-section, the chord dataset gathered by the chord scraper will be presented and evaluated. The performance and effectiveness of the chord scraper will also be evaluated.

When evaluating the chord scraper, it will be important to make a clear distinction between the quality of the input data versus the effectiveness of the chord scraper to correctly extract chords. For example, if a chord voicing contains notes that do not represent the associated chord symbol, it must be determined whether or not this is a cause of an incorrect MusicXML score, or inaccurate chord extraction by the Chord Scraper.

4.4.1 Jazz-Chords Dataset

To download the dataset, please go to (github link)

In order to gather the jazz-chords dataset, a chord scraper was developed and given as input a library of 171 fully arranged jazz piano solos. Details of the library can be found in section 3. The jazz-chords dataset consists of 7510 chord symbol and chord voicing pairs. All of the chord roots are transposed to C. There are a total of 23 different chord types present in the dataset. The dataset is presented as a data pickle, and in CSV format. The structure of the dataset can be seen below in figure 14. Figure 15 and 16 show the chord type distribution of the dataset.

|  |  |  |
| --- | --- | --- |
| ***Key*** | ***Data Type*** | ***Description*** |
| *type* | String | The type of chord, i.e., “major-seventh” |
| *extensions* | List of Dictionaries | A list containing any chord symbol extensions |
| *note\_numbers* | List of integers | A list containing chord voicings as note numbers |

Figure 14. The jazz-chords dataset

Chart, bar chart

Description automatically generated

Figure 15. A count plot showing the chord type distribution of the jazz-chords dataset.

|  |  |
| --- | --- |
| ***Chord Type*** | ***Count*** |
| dominant | 1875 |
| minor-seventh | 1569 |
| major | 928 |
| major-seventh | 818 |
| dominant-ninth | 359 |
| suspended-fourth | 322 |
| minor | 236 |
| minor-ninth | 206 |
| half-diminished | 204 |
| dominant-13th | 202 |
| major-sixth | 171 |
| major-ninth | 104 |
| minor-sixth | 83 |
| diminished-seventh | 82 |
| minor-11th | 75 |
| diminished | 70 |
| major-minor | 39 |
| augmented-seventh | 24 |
| major-13th | 12 |
| minor-13th | 11 |
| dominant-11th | 8 |
| augmented | 4 |
| suspended-second | 2 |

Figure 16. A table showing the chord type distribution of the jazz-chords dataset.

The chord distribution is typical of jazz piano, in which chord types such as dominant and major/minor-sevenths are most commonly used (citation).

The issue with the datasets chord distribution is that many of the chord types do not have enough associated chord voicings to be used in machine learning tasks. This means that in practice, the dataset can arguably only facilitate the generation of 4 different chord types – dominant, minor-seventh, major-seventh, and major.

As mentioned in section 1, Jazz musical theory defines a standardised set of possible notes that can be included in chord voicings. In order for the presented dataset to be of use, its contained chord voicings need to have a high degree of accuracy. In order to evaluate the level of accuracy in the dataset, the four most common chord types were tested for the presence of unwanted notes. Figure 17 shows which notes are unwanted for each of the chord types. Figures 18, 19, 20 and 21 show the note distribution of all of the chord voicings for each chord type.

|  |  |
| --- | --- |
| ***Chord Type*** | ***Unwanted Notes*** |
| Dominant | **-** |
| Minor-seventh | Db, E, F#, Ab, B |
| Major-seventh | Db, Eb, F, Ab, Bb |
| Major | Db, Eb, F, F#, Ab, Bb |

Figure 17. Most common chord types with their associated chord voicings unwanted notes

Chart, bar chart, histogram

Description automatically generated

Figure 18. A stacked bar-chart showing the total note distribution for all dominant chord voicings

Chart, bar chart, histogram

Description automatically generated

Figure 19. A stacked bar-chart showing the total note distribution for all minor-seventh chord voicings

Chart, bar chart

Description automatically generated

Figure 20. A stacked bar-chart showing the total note distribution for all major-seventh chord voicings

Chart, bar chart, histogram

Description automatically generated

Figure 21. A stacked bar-chart showing the total note distribution for all major chord voicings

As seen in each stacked bar-chart, some unwanted notes are present in each of the chord types. Figure 22 presents further chord voicing accuracy information.

|  |  |  |  |
| --- | --- | --- | --- |
| ***Chord Type*** | ***Total Notes*** | ***Total Unwanted Notes*** | ***Accuracy*** |
| Dominant | 8419 | 0 | 1 |
| Minor-seventh | 6999 | 258 | 0.963 |
| Major-seventh | 3587 | 79 | 0.978 |
| Major | 4259 | 406 | 0.904 |

Figure 22. A table showing chord the jazz-chords dataset chord voicing accuracy

As seen in figure 22, aside from the “major” voicings, the datasets chord voicings have a high degree of accuracy. In the following section, it will be determined whether the presence of unwanted notes is a result of the scrapers input being inaccurate, or a flaw in the chord scrapers functionality.

A total of 1872 chord symbol/chord voicing data pairs were removed from the raw dataset. This was due to the chord voicings having less than 3 notes. As explored in section 4.3.1, the chord scraper could not be designed in a way in which it could extract chord voicings which were arranged in a rhythmic pattern. It is suspected that this is the reason for so many chord voicings with less than 3 notes existing in the raw dataset. This will be further investigated in the following section.

4.4.2 Evaluation of Chord Scraper’s Performance

At the beginning of this section, the requirements of the chord scraper system were set out. Looking at the resulting dataset presented in the previous section, it can be clearly concluded that, for the most part, the chord scraper met all of its requirements. It was able to successfully extract pair of chord symbols and chord voicings. It was able to output both raw and refined datasets. The system was able to successfully scrape 166 unseen MusicXML files, having been designed using 5 test files. This suggests it will be able to successfully scrape further unseen MusicXML files. The system was also able to scrape all 171 MusicXML files in around 40 seconds. As mentioned in section 4.3, the chord scraper also features a CLI that makes working with the tool simple.

The “major” chord type voicings failed to meet the requirement of 97% accuracy. In order to find the cause of this inaccuracy, an investigation was conducted in which extracted voicings were manually compared with the MusicXML file from which they were obtained.

A total of 94 major chord pairs were randomly selected from the raw dataset for manual comparison. The results of this comparison showed that 100% of the inaccuracy was present in the MusicXML library, and thus existed before being presented to the chord scraper as input. This investigation suggests that the inaccuracy found within major chord voicings was a result of the inputted data, and not a result of the chord scrapers functionality. The high level of accuracy present in the other chord type groups also supports this suggestion. However, as the investigation was only based on a small sample of the total number of inaccurate chords, this suggestion cannot be fully asserted. If given more time, a larger sample could be examined.

A similar type of manual comparison was carried out in respect to the large number of chord voicings which had less than 3 notes. A total of 40 sub 3 note chord voicings were randomly selected from the raw dataset for comparison. The results of this comparison showed that all 40 voicings were arranged in a rhythmic pattern, and thus not all notes could be extracted by the chord scraper.

This investigation confirms a limitation of the designed chord scraper system which resulted in it not being able to successfully extract 20% of the chords present in the inputted MusicXML library. This issue has the potential to be resolved if given more time. One way in which this could be done was to find a way in which the chord scraper could distinguish between rhythmically and non-rhythmically arranged chord voicings. This would then allow for the *x-axis tolerance* to be adjusted accordingly.

5. Chord Generator

This section will detail the development of a novel conditional generative adversarial network (c-gan) with the aim of generating jazz chord voicings. This section will first look at how the jazz-chords data set was pre-processed and prepared for use as training data for the c-gan model. The section will then present some initial deep learning experiments that served as a precursor to developing the novel c-gan. The novel c-gan model will then be presented. Following this, the results of the c-gan will be presented and evaluated.

Sections 2.3, 2.4, and 2.5 and recommended precursors to this section. All models were trained using a Nvidia Tesla v100 GPU.

5.1 Objectives

- To prepare and encode the jazz-chords dataset for use as training data for deep learning models

- To experiment with different data encodings and different network architectures through carrying out some classification tasks

- To experiment with some existing c-gan models, as well as propose a novel c-gan model

- To report and evaluate the results of each experiment

5.2 Preparing the Training Data

Deep learning models require a large amount of data in order to be effectively trained. A c-gan model capable of generating jazz chord voicings needs to a large dataset of chord labels and chord voicings. For example, for the c-gan to generate a “major” chord, it needs to be given a large enough amount of “major” chord voicings for a mathematical relationship between label and voicing to be learnt. As mentioned, in section 2.5, c-gans are capable of generating multiple different types of output, meaning that they can be trained to generate multiple different types of chord voicings.

Looking at the chord distribution of the jazz-chords dataset, there are only 4 chord types that have enough associated voicings; these types are “dominant”, “minor-seventh”, “major”, and “major-seventh”. When working with labelled data, it is also good practice to ensure that each label is equally represented. In order to do this, 818 “dominant”, “minor-seventh”, and “major” chord pairs were randomly selected, and combined with the 818 “major-seventh” chord pairs to form an equally distributed dataset.

As presented in section 4.4.1, each of the chord type’s voicings contained a small margin of inaccuracy. This must be removed before the data can be considered for training the c-gan model. In order to do so, all of the unwanted notes seen in figure 17 were programmatically removed from all of their associated chord type’s voicings.

Furthermore, all of the chord voicing’s note distributions will be capped at the note F#4. This is so that the generator does not learn to generate chord voicings that will clash with their lead sheets melody. The resulting dataset is presented in figure 23, 24, 25, 26, and 27.

|  |  |
| --- | --- |
| ***Chord Label*** | ***Count*** |
| dominant | 818 |
| minor-seventh | 818 |
| major | 818 |
| major-seventh | 818 |

Figure 23. A table showing the chord label distribution of the training data

Chart, line chart

Description automatically generated

Figure

Chart, histogram

Description automatically generated

Chart

Description automatically generated

Looking at the stacked bar charts, all chord voicings now have no unwanted notes, and no not have notes that are F#4 and above.

The final step in preparing the training data is to represent the chord labels and chord voicings in a format that the c-gan can understand. This process is called data embedding or data encoding. Initially, a simple integer encoding will be applied to the chord labels; this can be seen in figure 28. The input layers of the c-gan may further embed these integer encodings.

|  |  |
| --- | --- |
| ***Chord Label*** | ***Integer Encoding*** |
| dominant | 0 |
| minor-seventh | 1 |
| major | 2 |
| major-seventh | 3 |

Figure 28. A table showing integer encoding for chord labels

In the jazz-chords dataset, the chord voicings are represented as a list of note numbers. The note numbers are numbers between 1 and 88, with each number representing a note on the piano. The number 1 is the lowest note on the piano. Two different chord voicing embeddings will be tested. The first will follow the approach set out by Chen et al. (2020) in their chord-jazzification system. Each chord voicing will be represented as a vector of 88 binary numbers, denoted as . Each binary number will represent a note on the piano, and the presence of a note will be known by a value of 1, rather than 0. Figure 29 highlights this embedding.

Chart, box and whisker chart

Description automatically generated

Figure 29. Chord voicing embedded as an 88 binary number vector

The second embedding approach has the intention of representing the chord voicings as binary images. This will allow for the experimentation of using convolutional layers (glossary) within the c-gan. On a piano, there is a repeating pattern of 12 notes, which is called an octave. By excluding the bottom 3 notes and top note, 7 full octaves remain. As none of the voicings within the training data contain these notes, they can be discarded. Thus, the voicings are embedded in 7x12 matrices. Figure 30 highlights this embedding.

A picture containing diagram

Description automatically generated

Figure 30. Chord voicing embedded as 12 x 7 binary number matrix

In order to determine which of these encodings better facilitates the mapping of a mathematical relationship with the chord labels, a simple classification task will be conducted. This is presented in the next section.

5.2 Classification task

As explained in sections 2.4 and 2.5, c-gan’s consist of 2 independent neural networks. The architecture of these networks depends on the type of data that is being generated. For example, in the task of image generation, convolutional networks are used for both the generator and discriminator (reference). If the c-gan is tasked with generating 1D vectors, then a simpler Deep Feedforward Network (DFN) (glossary) may be more suitable.

As presented in the above section, the chord voicings are encoded as both 1D vectors and as binary images. It is unknown which of these encodings better represents the ground truth of the chord voicings. In order to determine this, both data encodings will be used in a classification task experiment. To test the 1D vector encoding, a DFN classification network will be constructed. To test the binary image encoding, a CNN classification network will be constructed. Both networks will be optimised equally to try to ensure that the difference in results is influenced by the embedding approach.

By comparing the results of this experiment, the most effective chord voicing embedding approach can be determined.

5.2.1 Deep Feedforward Network

The initial network configuration was taken from a vanilla DFN found on the TensorFlow website (citation). Then, in order to find the ideal network architecture, Goodfellows principle was used – “The ideal network architecture for a task must be found via experimentation guided by monitoring the validation set error.”

The initial model featured an input layer of shape (None,88), and an output shape of (None,4), with only 1 dense layer in between. The output gave a prediction of which chord label the chord voicing input belongs too. For example, an output of [1, 0, 0,0] would mean that the model predicted the input as being a dominant chord.

To train the model, 2/3 of the dataset was given to the model over a number of training iterations called epochs. The remaining 1/3 of the dataset was then used to test the model’s accuracy in making predictions. The initial model had a test set accuracy of 60%, which seemed unusually low. The total loss of the model in respect to the test data was also high, at 0.6. A large total loss value means that inaccurate predictions are highly inaccurate, and usually indicates that there is some issue with the training dataset (reference). Common issues such as unclean or unbalanced data could be ruled out. However, a chapter in Goodfellow’s book suggested that high total loss in classification networks could be due two or more sets of labelled data having overlapping distributions to the point where the network struggled to tell them apart. By creating a scatter plot of the chord voicings note types, it could be seen that the “major” and “major-seventh” chord voicings had very similar distributions. This can be seen in figure 31.

Chart, scatter chart

Description automatically generated

Figure 31. A scatter plot showing major and minor note distributions of the datasets chord voicings

As “major-seventh” chords are much more frequent in jazz piano lead sheets, it was decided that the “major” voicings would be removed from the dataset. This can be seen in figure 32.

|  |  |
| --- | --- |
| ***Chord Label*** | ***Integer Encoding*** |
| dominant | 0 |
| minor-seventh | 1 |
| major-seventh | 2 |

Figure 32. Updated table showing integer encoding for chord labels

The removal of “major” chord voicings improved the models training and test set accuracy to 81% and 70% respectively. However, a significantly poorer test set accuracy suggested that the model was overfitting, meaning that its parameters were too specific to the training set, and not generalising the relationship between label and chord voicings.

In order to improve the accuracy of the model, further dense layers were incrementally added whilst observing the test set accuracy. To combat the issue of overfitting, a series of Dropout layers were added to the model. Dropout is a regularization technique that neutralises different parts of the network throughout training (reference). This prevents co-adaptation (glossary) and leads to a more balanced network that has a higher likelihood of generalising. The resulting DFN’s architecture is presented in figure 33.

Each layer had a Rectified Linear Unit (RELU) activation function, which outputs all negative values as 0 whilst leaving positive values unaffected. The model has a Sparse Categorical Cross-entropy loss function, which computes the cross-entropy loss between the labels and predictions. An Adam algorithm was used to optimize the network, which is a stochastic gradient descent method (reference). The loss function and optimizer are used to update the networks parameters in a direction which reduces the loss of the network (reference). The batch size of each training epoch was initially set to 32, and increased in powers of 2 to 1028, which was found to be optimal. Figures 34 and 35 show the accuracy and loss curves of training the network for 300 epochs. A regularisation principle called “early-stopping” suggests that training should be limited to the point at which the accuracy stops improving and/or the test data loss starts increasing. These two points indicate the beginning of the model overfitting.

Diagram

Description automatically generated

Figure 33. The DFN architecture used in the classification task

Graphical user interface, chart

Description automatically generated

Figure 34. The accuracy of the DFN after training for 300 epochs

Chart, line chart

Description automatically generated

Figure 35. The total loss of the DFN after training for 300 epochs

The final iteration of the model was trained for a total of 45 epochs. The results can be seen in figure 36.

|  |  |
| --- | --- |
| ***Test Accuracy*** | 0.8617 |
| ***Training Accuracy*** | 0.8631 |
| ***Test Loss*** | 0.3435 |
| ***Training Loss*** | 0.3280 |

Figure 36. The results of the final iteration of the DFN

This experiment has provided insight into the performance of representing the chord voicings as 88-note binary number vectors. The performance can now be compared with that of the chord matrices to determine which embedding to design the c-gan around. The experiment has also helped to develop a DFN architecture that can be used for both the discriminator and generator in the c-gan if it is decided that the chord voicings will be embedded as 88-note vectors.

5.2.2 Convolutional Neural Network

As mentioned in the previous section, CNN’s are used for machine learning tasks that involve images. They contain layers that apply convolution to the input images in order to extract patterns that can be used to learn unseen mathematical relationships between inputs and outputs (reference). The majority of previous research in the field of music and deep generation represents chord voicings as 1D arrays, like in the previous section. However, the presence of a repeating pattern on the piano makes the encoding of chord voicings as 2D arrays seem intuitive. The result is essentially a binary image; some examples can be seen in figure 37. By creating and optimising a CNN and performing a classification task, it will be determined whether or not this 2D encoding is more meaningfully understood by the computer than its 1D alternative.

Diagram

Description automatically generated

Figure 37. Chord voicings represented as binary images

The initial CNN architecture was based on AlexNet, a model proposed by Krizhevsky et al. (2012). The model is highly effective at image classification tasks. It is largely made up of pairs of convolutional and max pooling layers. The convolutional layers aim to expose patterns, and the max pooling layers aim to accentuate those patterns by maximising the highest pixel values within those patterns (reference). The output of the last max pooling layer is then flattened to a 1D vector and fed through a fully connected dense layer. This allows for a 3-neuron output layer, which indicates the model’s classification prediction.

As with the DFN described above, Goodfellows principle will be followed to improve the performance of the CNN through the guidance of the test set accuracy. It must also be ensured that the test set accuracy is not lower than the training set accuracy.

Some initial training runs showed that the model was overfitting to the training data, therefore a dropout layer was added after each convolutional + max pooling layer. The model’s loss optimizer, loss function, and batch size were the same as the DFN final model. The CNN model is presented in figure 38. Figures 39 and 40 show the performance of the CNN after training for 300 epochs.

Diagram

Description automatically generated

Figure 38. The CNN architecture used in the classification task

Chart

Description automatically generated

Figure 39. CNN accuracy after training for 300 epochs

Chart

Description automatically generated with medium confidence

Figure 40. CNN total loss after training for 300 epochs

Looking at the performance metrics, the network is optimally trained at around 200 epochs, after which the test set accuracy shows no improvement. The total losses also show no sign of reconverging, indicating that the model will continue over-fitting to the training data. The performance of the CNN model after 199 epochs is presented in figure 41.

|  |  |
| --- | --- |
| ***Test Accuracy*** | 0.8679 |
| ***Training Accuracy*** | 0.8771 |
| ***Test Loss*** | 0.3128 |
| ***Training Loss*** | 0.2653 |

Figure 41. Performance of the CNN model after training for 199 epochs

5.2.3 Results

The purpose of running the above classification tasks was to determine the best way in which to embed the chord voicings. Looking at figure 42 it can be seen that the binary image embedding, and CNN performed best. It must be noted however that there was no way to ensure that both networks were equally optimised.

The results suggest that representing chord voicings as binary images embeds a meaning closer to the ground truth, therefore that approach to embedding will be used in the c-gan model in the following section.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ***Metric*** | ***1D Vectors + DFN*** | | ***Binary Images + CNN*** | |
|  | ***Result*** | ***Difference*** | ***Result*** | ***Difference*** |
| Test Accuracy | 0.8617 | -0.0062 | 0.8679 | + 0.0062 |
| Test Loss | 0.3435 | -0.0307 | 0.3128 | +0.0307 |

Figure 42. A comparison of classification task results

5.3 Chord voicing generation

As described in section 2.5, c-gan’s can be conditioned to generate outputs on a given input. In the case of chord voicing generation, when given a chord label, a chord voicing that represents that label will be generated. In order for the generated chord voicings to have variance, a noise distribution is also passed into the model alongside the chord label. This means that points within that noise distribution will be conditioned on a particular chord label. By keeping the chord label fixed, and varying the point in the noise distribution, varying chord voicings representing that label will be generated.

This subsection will present the development of a conditional generative adversarial network (C-GAN) that is capable of generating accurate “dominant”, “minor-seventh”, and “major-seventh” chord voicings. The C-GAN should also be able to produce a good number of unique voicings. For example, if the model generated 100 “dominant” chord voicings, then at least 90% of those should be accurate, and 20% of those should be unique.

First, the process of building and optimizing the c-gan model will be described. Then, the results and capability of the developed model will be presented and evaluated.

5.3.1 Developing the conditional generative adversarial network (C-GAN)

5.3.1.1 cDCGAN

The model used to generate chord voicings was based on a class of CNNs called deep convolutional generative adversarial networks (DCGANs), first proposed by Radford et al. (2015). The model uses convolutional layers in both the generator and discriminator and specialises in image generation tasks. As the chord voicings are represented as binary images, the DCGAN is a highly suitable model. Brownlee (2019, p?) presents a working conditional DCGAN implementation built using TensorFlow; the source code he presented was copied and used as a basis for our model.

Brownlee’s DCGAN model was built to conditionally generate images of 10 different items of clothing. The model was trained using the MNIST fashion dataset (reference), which is made up of 28 x 28 grayscale images. The 10 different types of clothing had integer labels from 0-9. Figure 43 shows some examples of items from the MNIST dataset, as well as some conditionally generated items.



Figure 43. Real (top) and generated (bottom) images from the MNIST cDCGAN (Brownlee, 2019) (adapted image)

The MNIST generator took as input a vector of 100 values which represented a random noise distribution, as well as one of the clothing type integer labels. The generator then transformed and concatenated the noise distribution and label into a 3D matrix of size 7 x 7 x 256. This allows the generator to apply convolutions across the 128 stacked feature maps and select features that can be used to create a convincing 28x28 image. This process is typical of DCGAN generators and can be seen in figure 44.

Diagram

Description automatically generated

Figure 44. MNIST Fashion DCGAN generator (Brownlee, 2019) (Image adapted from Radford et al. 2015: 4)

The discriminator functions in a similar way to the CNN classification model proposed in 5.2.2. However instead of trying to predict the label of its input, the model tries to identify whether or not the input image is real (from the training data), or fake (from the generator). The MNIST cDCGAN takes as input a 28 x 28 image with its associated integer label and has an output layer consisting of a single sigmoid activated neuron. If the output is >0.5, then the discriminator thinks there is more chance that the input image is real, if it is <0.5, then the discriminator thinks there is more chance that the input image in fake.

The aim of the discriminator is to output values close to 1 for real images, and values close to 0 for fake images. The generator improves by moving its parameters in a direction that brings both of the discriminator’s outputs close to 0.5, meaning that it can no longer discriminate between real and generated images. This min max game can be expressed as:

Log(D(*x|y*)) is the discriminator's loss when estimating the probability that real data instance *x* with respect to label *y* is real. It can be thought of as the scaler that is outputted by the discriminator when inputted real images. D(G(*z|y*)) is the discriminator's loss when estimating the probability that a fake instance with respect to its label *y* is real. It can be thought of as 1 minus the scaler that is outputted by the discriminator when inputted generated images.

5.3.1.2 Adapting the cDCGAN model

The initial phase in adapting Brownlee’s MNIST cDCGAN model to the needs of this project was to change the model to fit the shape of the jazz-chords dataset. The first adaption was to change the input image shape of both the generator and discriminator from 28 x 28 to 7 x 12. Also, as the generator was built to generate images that are 28x28, the architecture of the generator needed to be adjusted. The generator was changed so that it concatenated the 100-vector noise distribution and integer label into a 7 x 1 x 129 3D array, and then applied 2D convolutions to output a 7 x 12. A sigmoid activation was also applied to the neurons of the output layer to ensure that the image pixel values were between 0 and 1.

5.3.1.2 Optimising the cDCGAN model

Using the adapted model, some initial training runs were performed using the following hyper parameters:

***Optimizer:*** Adam algorithm with learning rate = 0.0002

***Loss function:*** Binary cross-entropy

***Number of epochs***: 250

***Batch size:*** 20

After running several training runs, two issues arose. Firstly, the loss of the discriminator with respect to generated images quickly converged to 0 and stayed there. This indicated that the generator was not producing convincing images or improving over time. Despite tweaking the architecture of the generator model, this did not improve. Further research pointed towards the issue most likely being an issue with the training data (reference).

Looking back at figure 31, it was clear that each chord labels voicings had a distinct enough distribution. However, one issue that was identified was the fact that all of the notes of the chord voicings were distributed within only 3 of the 7 rows of the 7 x 12 matrices. This can be observed in figure 37 (p.x) and in the note distributions shown in figures 24, 25, and 27 (p.x) where C1-B1 and C5-B7 represent the 4 unused rows within the chord voicing matrices. It was theorised that this could be causing additional noise within the generator and subsequently causing the output binary images to not represent the training data distribution.

To combat this issue, each label’s 888 chord voicings were run through a function that removed the top row and last 3 rows, resulting in binary chord images that were of shape 3x12. This is visualised in figure 45.

A picture containing graphical user interface

Description automatically generated

Figure 45. Transforming chord binary images from 7x12 matrices to 3x12 matrices

After changing the shape of the chord images, the generator was updated to concatenate the noise vector and integer label into a 1 x 3 x 129 3D array, which then applied the same convolutions whilst up sampling the 2D axis to 3x12. This can be seen in the presented generator architecture in figure x.

This change in the image shape greatly improved performance of the adapted cDCGAN model, with the discriminator loss in respect to generated images stabilising at around 0.2 after 200 epochs.

However, this highlighted another issue in that the adversarial losses gave little insight into the real performance of the generator. This is a well-known issue within the literature, where recent works in deep learning literature have shown the benefits of also using perceptual metrics in improve the performance of C-GANS (Lucas et al., 2019: 1).

In order to implement some perceptual metrics, a function was created that checked the accuracy and uniqueness of each chord labels conditionally generated chords voicings after each training epoch. The function generated 100 chord voicings for each of the 3 chord labels. The accuracy was a measure of how many unwanted notes (as presented in figure 17, p.x) were present, and the uniqueness was implemented as 1 minus the number of chord voicings that were identical.

By tracking these perceptual metrics alongside the adversarial losses, Goodfellow’s principle could be more easily followed in order to optimise the network architecture. Through a process of experimentation, the adapted generator model was simplified to have only one 2D convolutional layer. Experimentation with the discriminator did not improve performance, and thus its architecture remained unchanged. The hyper parameters of the models, as shown previously, also remained unchanged. The final adapted cDCGAN generator and discriminator architectures are presented below in figures 46 and 47. The results of the model are presented in the next section.

Diagram

Description automatically generated

Figure 46. The adapted cDC-GAN Generator network architecture

Diagram

Description automatically generated

Figure 47. The adapted cDC-GAN discriminator network architecture

5.3.2 Results

After adapting Brownlee’s cDC-GAN model to fit the jazz-chords binary image dataset and implementing system to track measure perceivable metrics, Goodfellow’s principle was followed in order to maximise the accuracy and uniqueness of the generators output chord voicings. Figure 48 shows the accuracy and uniqueness of the generators output after 250 training epochs. The performance of the trained model is presented in figure 49. The adversarial losses of the adapted model after 250 training epochs are presented in figure 50.

Chart, histogram

Description automatically generated

Figure 48. Generated chord accuracy and uniqueness after 250 training epochs

|  |  |  |  |
| --- | --- | --- | --- |
|  | ***Dominant*** | ***Minor-seventh*** | ***Major-seventh*** |
| ***Chord Accuracy*** | 1.00 | 0.93 | 0.96 |
| ***Chord Uniqueness*** | 0.23 | 0.29 | 0.24 |

Figure 49. The performance of the trained adapted cDC-GAN model

Graphical user interface

Description automatically generated with medium confidence

Figure 50. Discriminator and Generator loss after 250 training epochs

As can be seen, the trained adapted cDC-GAN model is able to conditionally generate chord voicings of a given chord label with high accuracy and satisfactory uniqueness. After 250 epochs, the discriminators loss with respect to generated chords stabilised at around 0.2. This suggests that the generator performance is sub optimal, as it was only able to fool the discriminator 20% of the time. The discriminators loss with respect to real chords also stabilised at around 0.2, meaning it was able to correctly identify real chords in 80% of cases.

Some research from the cGAN literature suggests that adversarial losses are not always a meaningful indicator of the model’s performance, as there is no way to ensure that both the generator and discriminator models are equally performant (reference).

The adapted cDC-GAN model can now be used in the task of harmonic lead sheet arrangement, as it can take successfully take chord symbols and output chord voicings to represent them.

Figure 51 shows an example of an input and output of the trained generator model. The integer label of 0 represents a “dominant” chord, and the output matrix is a 3 x 12 chord voicing image. Each scalar represents a probability of that note being in the chord. Figure 52 shows the output chord matrix as note numbers and as note labels. Note that the first value in the output matrix is 16, or C2.

A picture containing table

Description automatically generated

Figure 51. An example of the trained generators input and output. The input is an integer label representing a “dominant” chord, and the output is a 3x12 matrices

Shape

Description automatically generated

Figure 52. The generated “dominant” chord as note numbers and note labels

6. Lead Sheet Arrangement System

The lead sheet arrangement system is the core system of this project. It takes as input a lead sheet in MusicXML format, and outputs an arrangement of that lead sheet. The lead sheet arranger system performs 3 main operations. First, it extracts the chord symbols from the lead sheet as well as some meta information. It then passes those chord symbols into the cDC-GAN generator model trained using the jazz-chords dataset. Finally, it takes the generated chords, and uses the meta information to format and insert them into the leadsheet, transforming it into a full arrangement. Figure 53 highlights these operations. In this section, the development of the lead sheet arranger system will be presented. Following this, some input and output examples will be presented, and those outputs along with the system itself will be evaluated.

Diagram

Description automatically generated

Figure 53. Lead sheet arranger system diagram

6.1 Development

The lead sheet arrangement system consists of three subsystems – *data\_extractor, chord\_generator, and leadsheet\_arranger*. Each of these systems function as individual components and serve to complete the 3 main operations outlined above. The following subsections will outline how each subsystem was developed.

6.1.1 Data-Extractor

The data extractor subsystem takes as input a MusicXML lead sheet, performs some data parsing and extraction functions, and outputs the lead sheets chord symbols and the lead sheets melody and meta-information. The purpose of the data extractor system is two-fold. Firstly, it extracts and presents the chords of the lead sheet to the chord\_generator subsystem in a format that the trained cDC-GAN generator can understand. And secondly, it passes the melody along with extracted meta-information such as the time signature & key signature to the leadsheet-arranger subsystem in a format that allows them to be transformed into a MusicXML full arrangement.

The development of the data-extractor system followed a very similar design pattern to that of the chord scraper (4.3.1). The system uses Python’s Element Tree to parse the lead sheet into a tree data structure format. By doing this, the chord symbols can be easily extracted by traversing through the tree and finding each MusicXML Harmony element (figure 12, p.x). The extracted chord symbols are transposed so that their root is C, and represented as chord labels, for example – “dominant” or “major-seventh”. This list of chord labels can then be passed to the chord\_generator subsystem.

The system also uses tree traversal to extract the lead sheets key signature (glossary), time signature (glossary), and divisions (glossary), as can be seen in figure 54. This meta information is stored in a Python dictionary and outputted to the leadsheet-arranger subsystem.

As mentioned above, the data-extractor system is very similar to the chord scraper system. This similarity meant that developing the data-extractor system involved reimplementing a lot of the chord scrapers functionality. This could have been avoided by using Python’s module importing feature to inherit or abstract some functionality. However, this would have created an interdependency between the systems, and caused issues if they were to be independently used in further research.

Diagram

Description automatically generated

Figure 54. MusicXML Ordered Tree Data Structure showing meta-information nodes

6.1.2 Chord-Generator

The chord\_generator system takes an input a list of chord labels, and outputs a list of chord voicings that represent the inputted chord labels. The chord voicings are outputted as a list of note numbers. For example, if the chord\_generator was given the following list as input:

[“dominant”, “major-seventh”, “dominant”]

The corresponding output would be something similar to:

[[16, 27, 32], [28, 32, 35, 39], [16, 27]]

The development of the chord\_generator system was straightforward. The system imports the trained cDC-GAN model and passes the list of chord labels into the model, which returns predictions. As shown in figure 51 (p.x), the predictions are 3x12 matrices. The chord\_generator system converts the matrices to the note number lists and outputs them to the leadsheet-arranger subsystem. In order to pass the chord labels into the generator model they are encoded as their associated integer number. The chord-generator system also generates vectors of 100 random noise scalars which are passed into the generator alongside the chord label integer notation.

As outlined in section 5, the trained cDC-GAN model can only generate chords for 3 chord labels – “dominant”, “minor-seventh”, and “major-seventh”. In order to increase the capability of the chord-generator system, each of the three compatible chord labels were paired with chord labels that were closely related. By doing this, the capacity of the chord-generators input was increased from 3 to 7. This was implemented by creating a dictionary of which the keys were the 7 chord labels, and the values were any of the 3 chord labels that the trained cDC-GAN model supported. The dictionary can be seen in figure 55.

chord\_types\_dict = {

"dominant": "dominant",

"dominant-ninth": "dominant",

"minor-seventh": "minor-seventh",

"minor": "minor-seventh",

"minor-sixth": "minor-seventh",

"major": "major-seventh",

"major-seventh": "major-seventh"

}

Figure 55. A Python dictionary showing how similar chord labels were assigned to 1 of the 3 cDC-GAN’s support chord labels

It must be noted that in doing this, some of the chords generated by the chord-generator would, by strict musical theory standard, be slightly incorrect. For example, although C minor-sixth and C minor-seventh chords share many of the same notes, a C minor-sixth chord has the note A in it. The note A would never in a “minor-seventh” voicing generated by the trained cDC-GAN model. The implications of this similarity dictionary will be explored further in the results section.

Aside from the 4 additional chord labels, there were many existing labels that the generator could not support. Each of these incompatible chord labels was dealt with by returning a very simple 2-note voicing consisting of the notes C2 and C3.

The implementation details of this substitution system were as follows. A function was created to validate each chord label before it was given to the cDC-GAN model. The function looked up each chord label in the *chord\_types\_dict*. If the label was present as a key in the dictionary, then the function simply returned the corresponding chord label value. However, if the chord label wasn’t present as a key, then a Key Error was raised. The function caught this error and printed an error message indicating that the chord could not be properly voiced. The simple 2-note voicing was then added to the list of generated chords.

6.1.2 Leadsheet-Arranger

The leadsheet-arranger subsystem contains the main method of the leadsheet arrangement system and is where all of the other subsystems are executed. The main methods of the other subsystems are imported using Python’s module import system. The leadsheet-arranger subsystem takes 3 inputs – leadsheet meta information, generated chord voicings, and the input leadsheet. It then uses the meta-information to transform the chord voicings into MusicXML elements and inserts them into the input lead sheet. The resulting output is an arrangement of the input lead sheet in MusicXML format.

The initial development of the leadsheet-arranger system focused on converting the generated chords into MusicXML elements that could be inserted into the leadsheet file. As the generated chords were all in the route of C, they needed to be transposed back to their original route. In order to do this, a function was created that simultaneously iterated through the leadsheet meta information and generated chord voicings. For each chord, the function would check the listed root note in the meta information, and then use a dictionary converter to transpose each note in the chord voicing the required number of steps. A use case of this is presented below:

For iteration *x*, the chord meta-information(*mi*), generated chord voicing (*v*), and transposed generated chord voicing (*tv)* are as follows:

*mi =* {'root': 'G', 'bass\_note': '','kind':'minor-seventh','extensions': []}

*v =* [28, 38, 43]

the root note ‘G’ is looked up in a dictionary to see how many notes away it is from C (shortest path either up or down). In this case G is 5 notes above C, meaning all of the notes in the chord voicing will be reduced by 5:

*tv =* [23, 33, 38]

The chord voicing is now rooted in its original note, as the note number 23 is equal to G2.

In order to convert the transposed chord voicings into MusicXML elements, each note number was converted into its corresponding note label. This process required the key signature of the lead sheet, as black notes can have 2 different note labels depending on what key they are in. An example of this would be the black note between the white notes C and D, which half of the key signatures would call C#, and the other Db. To implement the note integer-note label conversion, 2 dictionaries were created with note integers as keys and corresponding note labels as values. Using one of the dictionaries, the note integers could be converted. The advantage of creating look up dictionaries is they allow for each conversion to be performed in constant time (O(n)). An example transposed and note-labelled (*tvl*) can be seen below:

*tv =* [23, 33, 38]

*tvl =* [‘G2’, ‘F3’, ‘Bb3’]

To convert the note labelled voicings into MusicXML elements, a parser function was created that looped through each note label in each voicing and converted it into a matching MusicXML Note element. The Element Tree built in functions were used to create the Note elements. An example of a note label – Note element conversion is presented below:

*Note label =* ‘G2’

*Note element =* <note><pitch><step>G</step><octave>2</octave></pitch><duration /><voice>5</voice><type>half</type><stem>down</stem><staff>2</staff></note>

Once each chord voicing was represented as a series of MusicXML Note elements, they could be inserted into the correct position in the lead sheet. Each chord voicings position was determined by locating the bar of its associated chord symbol and inserting the notes at the end of that bar, resulting in them appearing in the bass clef. This can be seen in figure 10 (p.x), in which the notes in the bass clef are positioned directly after the last note in the treble clef.

Some error handling was also implemented within the leadsheet-arranger subsystem. For example, the system checks that the number of generated chords are equal to the number of chord symbols in the leadsheet. Checks are also performed on the lead sheet meta-information to ensure that it is accurate and conforms to MusicXML standards.

As the leadsheet-arranger system executes the main method of the leadsheet arrangement system, a CLI was developed to allow users to interact with the system and arrange lead sheets without having to open the source code. The CLI allows the user to specify an input directory and output directory. It also allows the user to enable a verbose mode, which results in errors being logged whilst the program is running. The CLI also has a help menu, which can be seen below in figure 56.

usage: leadsheet\_arranger.py [-h] [-v] [input\_directory] [output\_directory]

Leadsheet Arranger

positional arguments:

input\_directory path to input directory. Default is ./leadsheets/

output\_directory path to output directory. Default is ./arranged\_pieces/

optional arguments:

-h, --help show this help message and exit

-v, --verbose log parsing actions and errors.

Figure 56. Lead sheet arrangement system CLI help menu

When the leadsheet arrangement system is executed, it iterates through each file in the input directory, and executes on any files ending in “.MusicXML”. For each successful arrangement, a confirmation message is shown on the terminal along with the output directory.

6.2 Results and Evaluation

The lead sheet arrangement system was tested using the Wikifonia dataset, which is a collection of 7000 jazz lead sheets. Figures 57 and 58 show an example of the lead sheet arrangement system arranging a jazz lead sheet. In order to thoroughly test the system, it was executed with 200 randomly selected MusicXML lead sheets in the input directory. No major errors were reported, and 200 MusicXML arrangements were outputted. Of these 200 arrangements, 20 have been randomly selected for a technical evaluation, and 5 have been randomly selected for a participant led evaluation. Before presenting these evaluations, the capabilities and limitations of the whole system will be presented.

The system can arrange an infinite amount of lead sheets in MusicXML format. The arrangement of a standard-length lead sheet is performed in under 1 second.

The system can arrange any unseen lead sheet as long as it is in standard and correct MusicXML format.

In order to evaluate the effectiveness of the system, both a technical and participant led evaluation will be performed on 5 of the arrangements produced by the system.

The technical evaluation will take into account 3 factors – (1) Correctness – how many unwanted notes were present in the chord voicings. The accuracy of the cDC-GAN has already been analysed and presented in section 5.3.2, figure 49, however it will be briefly revisited here. (2) Playability – how easy or hard are the arrangements for a pianist to play. (3) Voicing Quality – how well are the chords voiced. Are the notes of the voicing in the best sounding part of the piano, is there any evidence of advanced voicing techniques such as split voicings.

For the participant led evaluation, 20 individuals with musical backgrounds will be asked to look and listen to 5 arrangements performed by the lead sheet arrangement system. They will asked to rate the quality of the arrangement through answering a set of 3 questions on a scale of 1-5.

QUESTIONS?

TABLE

Diagram, schematic

Description automatically generated

Figure 57. The first 8 bars of the leadsheet for the song Some Day My Prince Will Come

Diagram, schematic

Description automatically generated

Figure 58. An 8-bar arrangement performed by the Leadsheet Arrangement System on the song Some Day My Prince Will Come

Development of the system

Results- give an example input and output

What does it do? Can take any lead sheet in musicxml format, quick? Terminal menu?

Limitations – can only arrange 3 chord types + other similars, but some just cant

Doesn’t rhythmically arrange

Issues with odd number of chords in a bar

Conclusions/recommendations – summarises the contribution of the work and

identifies future work, etc. This is discussed in more detail below.

What can the results be used for? In other words, how can the reader benefit from

knowing what they have learnt from the report? To what can they apply your results?

n Can related research areas benefit from the results? Is what you have discovered or developed applicable to other research areas? What are the limitations of the approach

and of the results?

n Are the results theoretical or do they have a real-world application?

n How do the results compare with related research?

n Has the work identified new questions that need to be answered? Has the work identified

new areas of research?

Results

This paper was published in 2008 and presents a traditional programmatic solution to the task of lead sheet arrangement. The system takes as input chord symbols, and outputs a series of notes (voicing) for each of the chord symbols. The system is preprogramed with a set of jazz harmony rules, and voices each chord based on a series of conditional statements.

The resulting chord voicings were compared with *Band in a box (BIAB)* voicings in a percentual comparison test, in which the findings showed that the proposed system yielded a more preferable output.m

Although this system does not employ the use of deep learning, its results will be a useful benchmark for which to compare the results of this dissertation research project.

Some of the latter parts of the system also provide an aid on how generated chords may need to be adjusted to fit with the melody of the lead sheet.

For example, the presented system ensures that the highest note in the generated chord does not surpass the melody note that is played alongside it.

Table

Description automatically generated with medium confidence

Classification model notes

Represented the data in 3 different ways and used RNN and CNN to run classification tasks

Report results…

Look at where label classifications were different that expected -> are they in any way more accurate? Could this potentially be used to classify chords without chord symbols in XML files?? To gather more data??

GAN notes

**Embedding:** transform indexes into a vector of fixed size. E.g., representing words as numbers

**One-hot embedding:** create a “sparse” binary vector that is the length of the dictionary (100 words = 100 length vector). Then each word is represented by a 1 instead of a 0 in the vector.

\_\_\_\_\_\_\_\_\_\_\_

https://www.youtube.com/watch?v=BUNl0To1IVw&t=3018s:

Generative models:

Autoencoders and Variational Autoencoders (VAEs):

Learn lower-dimentional **latent space** and **sample** to generate input reconstructions

GANs:

Don’t explicitly model density, and instead just sample to generate new instances

When generator is trained – it is learning to transform from distribution of noise to target data disttibution

Gaussian noise 🡪 target data manifold

Z 🡪 Y

Take on point from latent noise distribution – will result in particular output in target data space.

The points in latent noise distribution represent different outputs

If we transverse in the Gaussian noise space we get interpellation of output (2 outputs mixed together)

55:10 – conditional GAN

Introduce pairs to G and D

Paired translation

12 vector integer notation -> 88 vector full representation

Effective distribution transformers

\_\_\_\_\_\_\_\_\_\_\_

**How c\_gan works exactly:**

The generator takes in a label and outputs a prediction

Label -> **Generator**  - > Fake Chord

Label + Fake Chord || Real Chord -> **Discriminator**  - > Real or Fake

The discriminator takes in pairs of labels + fake / real occurences and outputs a value on whether or not they are real

After the model is trained we give the generator labels and it gives us predictions

Adapted pix2pix model:

Transformed chord notes and label data into 7x12 matrices (omitting bottom 3 notes and top note)

Adapted pix2pix model to contain 1D layers as oppose to 2D layers (see paper)\

Initial results:

The loss of the discriminator on both real and generated data rapidly went to near zero, why? Probably the generator

**Novel model**

“Both G and D could be a non-linear mapping function, such as a multi-layer

perceptron.”

**Discriminator**

Discriminator that takes in concatenation of labels and either fake/real chords

How should this data be represented/concatenated?

Look at results of classification tasks

When concatenating label and chord, should an extra dimension be added?

Discriminator should have a single output with sigmoid activation (value between 0 and 1 indicating whether or not it is fake/real)

If using 1d representation, should concatenation be on a third axis, i.e. (88,) + (88,) 🡪 (1, 176) **OR** on the x axis, ie. (88,) + (88,) 🡪 (176,)

How can this decision be made? Other than just testing it, is there some reasons for why one would perform better than the other?

Probably the third axis, as it represents the relationship between label and chord in a more meaningful way – EXPAND ON THIS

If using 2d representation of labels/chords -> what is the use of convolutions? Can they actually find patterns -> research this

**Generator**

Takes in a label and transforms it into a fake chord candidate

Again, do we need to use convolutional layers? Or will deep/fully connected layers do the job

Real examples = (label + real chord) + [1] (label for discriminator)

Fake examples = (label + fake chord) + [0] (label for discriminator)

**Training**

**GAN Loss Function:**

the generator tries to minimize the following function while the discriminator tries to maximize it:

* D(x) is the discriminator's estimate of the probability that real data instance x is real.
* Ex is the expected value over all real data instances.
* G(z) is the generator's output when given noise z.
* D(G(z)) is the discriminator's estimate of the probability that a fake instance is real.
* Ez is the expected value over all random inputs to the generator (in effect, the expected value over all generated fake instances G(z)).
* The formula derives from the [cross-entropy](https://developers.google.com/machine-learning/glossary#cross-entropy) between the real and generated distributions.

(<https://developers.google.com/machine-learning/gan/loss>)

**Conditional GAN Loss Function**

**Backpropagation**

**Generator**

**Results:**

**Data Embedding**

I experimented with the data embedding:

**1)** 2 different 1D representations:

**1.** Source chord, real chord = (12,)(88,)

**2.** Source chord, real chord = (88,)(88,) [source chord(label) embedded to full 88 note vector in the middle C octave

**2)** a 2D representation:

(88,)(88,) 1D embeddings reduced to (84,)(84,) by removing bottom 3 notes and top note. Then transformed to (7,12)(7,12)

**Classification Models**

In order to determine the best embedding for the GAN, as well as the best type of network for the discriminator and generator, performed some classification tasks using all the embeddings and a combination of RNN and CNN.

1D RNN classifier using both **1.1 and 1.2** (above)

2D CNN classifier using **2**

Results:

Both performed incredibly well:

Give functions, data, graphs, etc

**Model 1: 2d pix2pix model (adapted for my dataset):**

Results showed that the discriminators loss quickly went to near zero – meaning that the generator was not generating convincing chords from source chords.

Show graph of loss over time and any other graphs/chards that were in 1D paper or other papers

Show some example generated chords

Why?

Unclear, using convolutional layers on simple binary chord matrices potentially didn’t yield any meaningful patterns

patchGAN uses a segment of the real/fake images (chords) passed in. ie. If matrices is 128x128, it looks at a 16x16 segment or multiple segments

this approach wont be as effective for chord data, as there would be notes spread around the vector, particularly on the x-axis (0 axis??)

Also mention loss functions /learning rate etc

**Model 2: 1D pix2pix model (adapted for my 1D dataset):**

Results?

Same as above

**Model 3: 1D Novel Model**

This model was born out of some initial testing using novel classifier networks

**V1**

The discriminator had a single output neuron with sigmoid activation with binary-crossentropy loss function

Results…. Loss of the discriminator on fake chords quickly reduced to 0, meaning generator was not improving

\*mention the loss function of the GAN here\* - explain it etc

\*more results here with graphs etc and chords shown in images etc with stats\*

**V2**

Created an 88 neuron output layer on the discriminator

\* mention how this changes the way the loss function works in the c\_gan model

Discriminator loss on real chords went from 1.5 – 1.2 over 11 thousand iterations of training

Loss on fake chords quickly went to zero

**Changed way source chord and real/fake chord concatenated**

Initially it was (88,)(88,) 🡪 (2, 88)

Maybe dense layers don’t perform as well on matrices?? Reference/research

Changed to (88,)(88,)

**Leadsheet XML Arranger**

**INPUT:** leadsheet (musicxml)

Step 1. Extract chord symbols

Step 2. Embed chord symbols as 88 “source-chord” note vectors (see preprocessing chapter)

Step 3. Pass source chord vectors into c\_gan model

Step 4. Combine leadsheet melody with generated chords into full arrangement

**OUTPUT:** Chords + Melody Arrangement

Data parsed from lead sheets

**Meta information:**

-Key

-Time signature

**Chords:**

-Chord symbol

-Bar number

-Position in bar

-Note at chord position (won’t account for 2 voicings / not sure which note it will get if chord present)

Data Issues

The number of different types of chords in the data was skewed :

Insert pandas table

1000 dominant 7 wheras much less of other chords

Data skewness

Data should be bell curve

But currently it is right skewed

Something to do with mode median

SOLUTION?

Get more data + remove some label occurences

WHEN SPLITTING TEST vs TRAIN:

INITIAL DATA COUNTS WITHOUT WIDENING CHORD SELECTIONS:

dominant 1177

minor-seventh 1053

major-seventh 674

major 453

dominant-ninth 255

suspended-fourth 249

dominant-13th 159

minor-ninth 133

half-diminished 129

minor 117

major-sixth 96

major-ninth 68

minor-11th 63

diminished-seventh 59

diminished 44

minor-sixth 38

Name: label, dtype: int64

Performing data analysis using pandas and matlabplot

Removed all chord labels that have less than 600 occurrences

Created a stacked bar chart representation of each label to analyse data and note occurrences

\*\*insert bar charts\*\*

For the large part the data was good - \*\*insert\*\* margin of error numbers

However a small number of labels had wrong notes associated with them

\*\*insert\*\* some examples with numbers of wrong notes

Therefore further data cleaning was performed. Created array of unwanted notes for each label, and iteratively removed those notes from each chord vector.

\*\* insert cleaned bar charts\*\*

We now have

References

Ji, S., Luo, J. and Yang, X., 2020. A Comprehensive Survey on Deep Music Generation: Multi-level Representations, Algorithms, Evaluations, and Future Directions. arXiv preprint arXiv:2011.06801.

Hadjeres, G., Pachet, F. and Nielsen, F., 2017, July. Deepbach: a steerable model for bach chorales generation. In International Conference on Machine Learning (pp. 1362-1371). PMLR.

Mirza, M., & Osindero, S. (2014). Conditional generative adversarial nets. arXiv preprint arXiv:1411.1784.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A. and Bengio, Y., 2014. Generative adversarial nets. Advances in neural information processing systems, 27.

Szelogowski, D., 2021. Generative Deep Learning for Virtuosic Classical Music: Generative Adversarial Networks as Renowned Composers. arXiv preprint arXiv:2101.00169.

Zhao, Y., Qiu, L., Ai, W., Shi, F. and Zhu, S.C., 2020. Vertical-Horizontal Structured Attention for Generating Music with Chords. arXiv preprint arXiv:2011.09078.

Szelogowski, D., 2021. Generative Deep Learning for Virtuosic Classical Music: Generative Adversarial Networks as Renowned Composers. arXiv preprint arXiv:2101.00169.

Dong, H.W., Hsiao, W.Y., Yang, L.C. and Yang, Y.H., 2018, April. Musegan: Multi-track sequential generative adversarial networks for symbolic music generation and accompaniment. In Thirty-Second AAAI Conference on Artificial Intelligence.

Angioloni, L., Borghuis, T., Brusci, L. and Frasconi, P., 2020, November. Conlon: A pseudo-song generator based on a new pianoroll, wasserstein autoencoders, and optimal interpolations. In Proceedings of the 21th International Society for Music Information Retrieval Conference ISMIR MTL2020 (pp. 876-883).

Valenti, A., Carta, A. and Bacciu, D., 2020. Learning a latent space of style-aware symbolic music representations by adversarial autoencoders. arXiv preprint arXiv:2001.05494.

Engel, J., Resnick, C., Roberts, A., Dieleman, S., Norouzi, M., Eck, D. and Simonyan, K., 2017, July. Neural audio synthesis of musical notes with wavenet autoencoders. In International Conference on Machine Learning (pp. 1068-1077). PMLR.

Isola, P., Zhu, J.Y., Zhou, T. and Efros, A.A., 2017. Image-to-image translation with conditional adversarial networks. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 1125-1134).

Good, M., 2001. MusicXML for notation and analysis. The virtual score: representation, retrieval, restoration, 12(113-124), p.160.

Lucas, A., Lopez-Tapia, S., Molina, R. and Katsaggelos, A.K., 2019. Generative adversarial networks and perceptual losses for video super-resolution. IEEE Transactions on Image Processing, 28(7), pp.3312-3327.