Data Representations in Neural Network Based Chord Progression Generation Methods

Felix Wu

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2020

Abstract

There are many ways in which one can represent music as data, and choosing the right one is an important step in maximising the performance of a computer model in the musical domain. Many researchers in the field of music generation choose a representation without fully considering all possible options, as there is little work done in comparing them. In this work I investigate the effects of various chord representations for the task of chord generation, and propose two novel representations, Normalised Note Encoding and Two-Hot Encoding, to improve on existing ones currently used in the literature. I found that my representations provide a 1-2% improvement over some others when applied to a variety of models such as Feed-Forward, Convolutional, and LSTM neural networks. However, the One-Hot representation was consistently the best performing in all situations, even though it is arguably the simplest, so I highly recommend that this representation is used where possible. I also propose a new evaluation metric for the performance of these models, based on a similarity calculation using word embeddings, which intends to provide a more human-like evaluation over a simple black-and-white metric.

April 24, 2020

Acknowledgements

Thanks to my supervisor Hiroshi Shimodaira for guiding me through the machine learning part of this project and giving me valuable feedback on the report.

Thanks especially to James Owers for taking time out of his PhD to help guide me through the planning process, provide valuable suggestions on the bridge between music and machine learning, and give excellent feedback on the report. I have learned a lot through James and I really appreciate what he has done to help the project come together.

Thanks also to my friends and family who helped with proof-reading the report.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

Computer-generated music is a fascinating topic, partly for the potential of creating new and interesting ideas, and partly for the excitement that is the application of new technologies to a task that one traditionally might have thought was reserved exclusively for humans.

While the task of creating fully-featured songs is still some distance away for many genres of music, many have explored automation of smaller parts of the music creation process, leaving the rest for the musician to finish. One such part is the chord progression, which can be thought of as the harmonic blueprint of a song, and can serve as the foundations upon which other melodic content is built.

With more research into music generation, better algorithms can be created to provide two main things: we can use them as a tool to come up more varied and creative ideas, and we can adapt them to use as an educational tool if the algorithm is genre-specific.

1.2 Related Work

Automatic chord progression generation is not a new topic by any means. The application of formal linguistical techniques to music in general can be seen as early as 1968 by Winograd [39], and for chord progressions in 1984 by Steedman [32] and in 1991 by Johnson-Laird [13]. Rule-based systems [8, 26] and stochastic mathematical models such as Markov Models [1, 28, 41] were very popular in the 80s and 90s, as well as more advanced methods like Genetic Algorithms [12, 22, 36].

Today, the application of Artificial Neural Networks for chord generation is becoming more and more prevalent. Networks such as Recurrent Neural Networks (RNNs) [25], and Generative Adversarial Networks (GANs) (Recurrent [24] and Convolutional [40]) have proven themselves as viable options in this domain.

However, by far the most common and successful network for chord generation seems to be Long Short-Term Memory Neural Networks (LSTMs). Eck et al. [9] improves on the performance of standard RNNs by using an LSTM to generate Blues chord progressions, Choi et al. [5] generate chord progressions using an LSTM from a textual representation, Lim et al. [19] present a Bi-directional LSTM for generating a chord sequence from a symbolic melody, and Mao et al. [21] present DeepJ, a MIDI generating LSTM with a parameter to allow choosing between different styles of music.

There are many ways in which these authors choose to represent their data before feeding it into their models, even for very similar tasks, which makes it unclear if there are representations that should only be used in certain situations or if there are some that should be avoided entirely. As far as I know, there is currently no work done on comparing different representations, which is the gap I intend to fill with this work.

1.3 Goals and Contributions

The focus of this work is to improve on the accuracy of neural network models for the task of chord generation using different kinds of input data representation, and to compare the differences in performance for existing representations. Unless stated otherwise, I refer to chord generation specifically as: *The prediction and generation of a chord following a fixed-length sequence of chords* (Figure 1.1).

I propose two novel chord representations and compare the differences in performance in this task against other representations used in the available literature. I have applied the representations to a Feed-Forward Neural Network (FNN), a Convolutional Neural Network (CNN), and a Long Short-Term Memory Neural Network (LSTM) to investigate their effects on a range of models commonly used for this task.

I also propose a novel way to evaluate the generated chord progressions, which uses word embeddings [23] to calculate the similarity between the generated and expected chord. I provide theoretical arguments and preliminary evidence to show that word embeddings are a better form of evaluation than existing, accuracy-based ones.

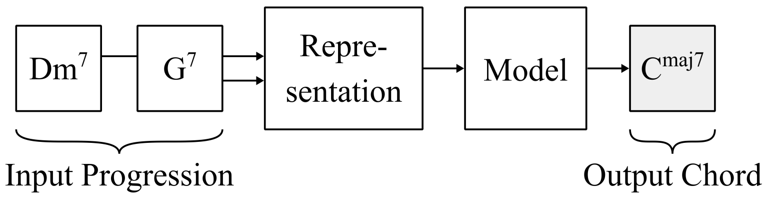

Figure 1.1: The task being considered in this work.

1.4 Report Structure

I start with introducing some basic music theory concepts, which should be enough to understand the rest of the project in chapter 2.

In chapter 3, I do a more in-depth literature review, as well as introducing my contributions on a more technical level and relative to what is lacking in the literature.

The dataset, why I chose it, how I obtained, and how I reprocessed it is described in chapter 4.

In chapter 5 I describe in detail how my experiments are implemented, and why I chose to run them. I discuss what I expect to happen based on what I see in the literature, as well as my own judgement.

The results for the experiments are shown in chapter 6, where I interpret them based on what I expected to see, and if I did not expect to see the results, I provide potential explanations as to why.

Finally, in chapter 7, I compile all of my findings to provide a final discussion on the main takeaways from my results, with a short discussion on what future work is possible from here.

Chapter 2

Music Theory Basics

First of all, I will cover some music theory concepts and terminology that I think is necessary for this work. I may make some very slight simplifications where I think it is appropriate.

2.1 Notes and Equal Temperament

Almost all modern western music uses the *12-tone equal temperament*, which is a system that determines the distance between two adjacent notes. It divides an *octave*, which is an interval between two notes that have a 2:1 frequency ratio, into 12 equal parts such that the ratios between all adjacent notes are equal. This particular ratio comes out as $\sqrt[12]{2}$, and two notes that have this frequency ratio are a *semitone (st.)* apart.

Each note in an octave has its own name which is made up of a letter between A and G followed optionally by a *flat* (♭), *sharp* (♯), or *natural* (♮) symbol (Table 2.1). Some notes may have multiple possible names depending on the musical context. Notice that after 12 semitones we return to the same note name that we started with, repeating the cycle.

We also have names for all the intervals between two notes just like the name *octave* (Table 2.2). There are also more names for intervals greater than 12 semitones like *Minor Ninth* (13 st.) and *Major Ninth* (14 st.), which follow a very similar pattern.

Semitones Above C	0	1	2	3	4	5	6
Note Name	C	C♯ / D♭	D	D♯ / E♭	E	F	F♯ / G♭
Semitones Above C	7	8	9	10	11	12	
Note Name	G	G♯ / A♭	A	A♯ / B♭	B	C	

Table 2.1: Note Names in an octave with equal temperament

Semitones	Interval Name
0	Unison
1	Minor Second
2	Major Second
3	Minor Third
4	Major Third
5	Fourth
6	Augmented Fourth or Diminished Fifth
7	Fifth
8	Minor Sixth
9	Major Sixth
10	Minor Seventh
11	Major Seventh
12	Octave

Table 2.2: Interval names for each semitone between two notes.

Notes	Chord Name
C, E, G	C major (or just C)
C, E♭, G	C minor (or Cm)
A, C, E	A major (or A)
A, C, E, G	A seven (or A^7)
A, C, E, G♯	A major seven (or A^{maj7})

Table 2.3: Chord Name Examples

2.2 Chords and Chord Progressions

Chords are a set of notes, usually three or more, that are played at the same time. But for most intents and purposes, notes that are played in close proximity to each other can also be considered a chord. Each chord has a name that describes the notes that it represents, table 2.3 shows some examples of chord names. The note from which the chord takes its name, usually the lowest note, is called the *root* note.

A *chord progression* is a series of chords that act as the blueprint or harmonic foundation of the melodic content in a song. It is not strictly necessary for the chords to be played, since they could be inferred from the melodic content, however, the chord progression still exists on a theoretical level. The chords in a progression can each last for any amount of time, but are usually similar in length throughout the song. Chord progressions are also often repeated throughout the song sometimes with slight variations.

2.3 Key

Every song has a *key*, a set of notes (like a chord) that best represents the harmonic content of the entire song. The chord that the notes in the key make is called the *tonic*, and is the chord that

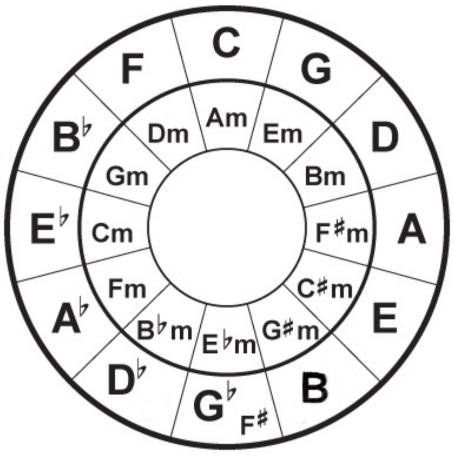

Figure 2.1: The Circle of Fifths

provides the biggest sense of arrival or rest. Other chords provide a sense of *tension* of varying degrees, and one usually expects the tension to eventually be *resolved* back to the tonic. When identifying the key of a song, a good start would usually be to look at the first or last chord.

The *Roman Numeral Notation* is a way of writing chords relative to the key of the songs. This allows two identical songs in different keys to share the same chord notation. For example, a ”D” chord in the key of ”C” would be written as ”II” and ”G” as ”V”.

2.4 Circle of Fifths

The *Circle of Fifths* (Figure 2.1) is a graphical representation of chord relationships where the major and minor chords in all 12 tones are laid out in a circle such that each neighbouring chord is a fifth away. When looking for the next chord in a chord progression, choosing nearby chords on the circle tend to sound better than those further away.

2.5 Other Terminology

- Bar: a measure of time, in this case the length of four *beats*
- Transposition: to transpose notes is to change their pitch up or down
- Harmonic function: the harmonic function of a chord in a progression expresses what the role of the chord is in the context of the progression. Sometimes, chords with very similar notes can have a very different function within the piece.
- Triad: a chord that consists of only the root, the third (minor or major), and the fifth (in no particular order).

Chapter 3

Background

3.1 The Link to Natural Language Processing

The problem of chord progression generation is, in many ways, similar to the problem of sentence generation in Natural Language Processing. Words are to sentences what chords are to chord progressions, and the context and order of the words/chords will change the meaning of the sentence/progression. The question ”Given these five *chords*, what is the next chord in the *progression*?” is a similar question to ”Given these five *words*, what is the next word in the *sentence*?” and can be answered in a similar way using methods from NLP.

One of the most common ways to represent words in NLP is the one-hot encoding. This is where the vector has a position for each unique word in the dataset, and all values are 0 except the represented word, which is 1. Because of this analogy, we can use one-hot encoding to represent the chords in the exact same way, treating each unique chord as its own independent word.

However, if you look at how letters relate to words and how notes relate to chords, the analogy for words and chords breaks down. The notes in a chord have a much greater say in the meaning of a chord compared to the letters in a word. Just as an example, adding the note B♭ to a C major chord (notes: C, E, G), turns it into a C^7 chord, which is different but still strongly related. Adding the letter *e* to *hat* makes *hate*, which is anything but related. This is why some authors choose to represent their chords in a way that captures the information of the notes within the chords, in the hopes that this added information could increase the performance of their models.

3.2 Criticism of Related Work

In this section I go into more detail about why my goals are what they are in relation to what is missing from the literature.

3.2.1 Lack of Representation Comparisons

Representing data in different ways can help your models better understand the data it is given. They need to be meaningful and strike a balance between simplicity and information retention. There are three main families of representation:

- **One-hot encoding:** does not contain any information about context or content;
- **Many-hot note encoding:** contains information about the content of the chord, i.e. the notes of the chord;
- **Chord embedding:** a vectorisation model like Word2Vec is trained to produce a representation where the cosine similarity of two similar chords is large.

From the authors whose work involves chord generation, most choose to represent their chords as one-hot vectors [4, 5, 6, 19] or many-hot vectors [9, 25] for their neural networks. But, surprisingly few authors use chord embeddings for chord generation [3], despite the success of word embeddings in other non-musical domains.

Madjiheurem et al. [20] present work on exploring different types of chord embeddings, although not specifically for chord generation.

It is clear that there is an interest in chord representations, but it is not clear if there is an all-round best choice, or if some should be avoided entirely. With the exception of Chord2Vec [20], existing works do not to explore the effects of other representations for their models, so it is not generally known which representations work better and in what contexts. My goal of exploring the effects of different representations will hopefully contribute to better understanding in this particular area.

3.2.2 Issues with Objective Evaluation

A straight-forward, accuracy-based evaluation could be considered: *”is the generated chord the same as the chord I expected?”*, where the input progression is a subset of N chords from an unseen chord progression (the same N chords as in the formulation from section 3.1), and the expected chord is the chord that comes next in that progression. Answering *yes* to this question can be represented with a score of 1 and *no* with a score of 0, and the average score of all validation examples would represent the accuracy of the model.

However, this is far from how humans evaluate chord progressions in reality. **If the generated chord is not the expected chord but is functionally similar, we might like to give it a score between 0 and 1, rather than a straight 0.** There is a certain flexibility in human evaluation that the accuracy-based method cannot capture. Moreover, there will inevitably be multiple answers accepted by even the same expert musician, so it is definitely not acceptable to model this problem as a single-solution task.

Many authors use this accuracy-based evaluation method, which means their methods are prone to this issue of inflexibility. Mogren [24] uses different metrics for evaluation of the MIDI output based on polyphony, scale consistency, repetitions, and tone span, which has given more insight into the quality of the generated music, but is still based on an uncompromising accuracy-based evaluation.

3.2.3 Issues with Subjective Evaluation

If we want a human-like flexibility in our evaluation, then why not use real humans in a subjective user-preference study? I believe that unless you have a very high quality and sizeable set of participants, i.e. they are an expert in the genre, the results would be too variable to be useful, leaving a lot to be desired.

Examples from the literature:

- Chen et al. [4] conduct a user study with 106 participants to evaluate their LSTM and CNN networks for folk music melody and chord generation. However, the only statistic they provide regarding the background of their participants is that 69.81% have *"experience in music"*. It is unclear exactly how experienced the participants are or if they are even experienced in folk music at all.
- Chu et al. [6] conduct a user study with 27 participants on pop music generation. Again, there is little information on the musical backgrounds of the participants, only that they are university students.
- Mao et al. [21] conduct a user study with 20 participants on classical music generation, with no information on the musical backgrounds of the participants.

It is true that many people listen to music regularly, so one could argue that it qualifies them to be a judge in evaluating generated music. But being a consumer of music does not mean you understand the intricacies of a specific genre like folk or classical, or music theory in general, and so the generated music will likely be indistinguishable from the real music to the untrained ear.

3.3 Proposed Similarity-based Evaluation Metric

To overcome the issue of inflexibility as discussed in section 3.2.2, while avoiding the need to use expensive and unreliable user-preference studies, I propose to evaluate the generated chords based on a similarity calculation.

In NLP, word embeddings [23] are often used to capture similarity between words, which is achieved by encoding each word as a vector such that two words with similar meanings have similar vectors. This requires a separate neural network to be trained on the dataset, which can determine the similarity between words based on the contexts in which that they are used. Once we have a way to convert a word into a vector, the similarity of two words can be calculated quantitatively using the cosine similarity of their vectors.

A similarity-based metric like this is exactly what we need to get closer to human-like chord evaluations, since it allows for the generated chords to be scored with a value between 0 and 1. More precisely, the similarity score for a model can be calculated as the average cosine similarity between the expected and generated chords in an unseen validation set.

I believe this evaluation metric is more representative of the performance of a model than the accuracy-based metric, simply because it behaves more like a real human. It is able to give a higher score to answers that are more accepted, and because of this, is also capable of handling situations in which there are multiple correct answers.

It is hard to prove this empirically, so I have shown both the accuracy-based and similarity-based evaluation scores in all experimental results in this work.

Chapter 4

Dataset

4.1 Genre of Choice: Jazz

I believe that it is important to stick to one genre of music when creating the dataset. Otherwise the model could struggle to produce suitable chords as it would have the additional task of identifying the genre before even generating the chord itself.

With this in mind, I chose the genre of Jazz because the chord progressions in Jazz have an importance and complexity unrivalled by the progressions of any other genre. However, the dataset could also be replaced with songs from another genre and still apply in the same way.

Jazz is inherently a very improvisational genre of music, so a performer would think of a Jazz song mainly in terms of its chord progression. Because of this, Jazz artists are always looking to create new and interesting progressions to stand out. Possibly the most famous chord progression in Jazz is *Giant Steps* by *John Coltrane*, which has musicians all over the world learning to improvise over it ever since it was released in 1960.

There is a well established branch of music theory dedicated to Jazz chords and harmony [37], and it is even sometimes difficult to talk about some non-Jazz songs without borrowing a few concepts from it.

Of course, within Jazz there are different styles too, but narrowing down the genre even more would decrease the size of the dataset, so I believe choosing Jazz as a whole is a good compromise.

4.2 Source

Since the dataset contains only a single genre, it is also important to choose songs that are representative of said genre. Wikipedia maintains an impressively comprehensive list of Jazz standards [38], which is a good representation of what is considered to be Jazz. There is also a dedicated website and community on *jazzstandards.com* for listing and ranking the most commonly played jazz songs.

The dataset used in this work comes directly from *iReal Pro* [34], a forum and mobile app for sharing and playing chord progressions in various genres of music. The main purpose of the app is to play back a song's backing track, complete with drums, piano, bass and more, depending on the song, so that users can practise playing their instrument on top. The forum serves as a platform for users to share these backing tracks, and includes a few large officially-maintained compilations.

A song is defined in the app mainly by its chord progression, along with some basic header information such as tempo and style of drumming, as these are the only things it needs to automatically generate the backing track. Conveniently, the app supports a feature to export the backing tracks in *MusicXML* format, which is a common XML-based file format for western musical notation.

The most common genre available on iReal Pro is Jazz; the biggest of the official compilations on the forum is a collection of 1350 Jazz songs, with hundreds of other Jazz songs shared by the users of the platform. Most of these songs can be found on the Wikipedia list of Jazz standards [38] or *jazzstandards.com*, and are what I use to create my dataset.

There are other larger and more famous music datasets available, however, I could not use them for various reasons. The Million Song Dataset [2] for example, is extremely popular, however, it does not actually have information about the content of the songs, but rather metadata such as genre, loudness, duration etc. There are also many datasets with MIDI transcriptions of songs, such as The Lakh MIDI Dataset [29], however, with no explicit chords transcribed, I would need to guess the chords form just the MIDI notes, which is an entire research area in itself. The Bach Chorale Harmony Dataset [7] does have these chords explicitly transcribed, but I want to stick to the genre of Jazz for its rich harmonic content.

4.3 Preprocessing

The dataset I have is in MusicXML format, making it easy to parse and extract the chord progressions into a more useful format. The XML structure is shown in Table 4.1, describing where and what various pieces of information are.

The MusicXML data was parsed to a modified version of the format presented by Harte et al. [11] (Figure 4.1), which defines a chord using three parts:

- A *root note*.
- A list of *degrees*, called *components*. A degree is a note from the chord, represented by the number of semitones that the note is above the root note of the chord.
- A *bass note*, which is also a degree. The bass note is a way of transposing one of the notes in the chord down an octave, which changes the sound of the chord (*voicing*) when played, but doesn't change the abstract concept or harmonic function of the chord.

In this work voicings were ignored because they do not have as much impact on the harmonic function of the chords on a theoretical level. Because of this, the bass notes are ignored if they are just changing the voicing and not adding new notes. So for C major (C, E, G) if the bass note is a G it is ignored because it just changes the voicing, but if the bass note is a B, it is added to the components.

Each chord is also normalised relative to its key, so instead of storing the note name, the number of semitones it is above the key root is stored. This is because the key itself does not have any harmonic relevance, it just specifies at what pitch the song should be played at.

The textual representation of this format is `root:[degree,degree,...]`. For example, a C major chord in the key of B is represented as `1:[4,7]` because C is 1 semitone above B, E is 4 semitones above C, and G is 7 above C.

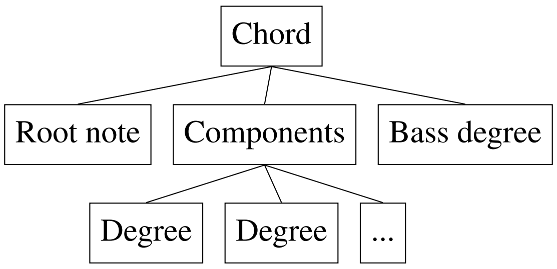

Figure 4.1: Chord Structure Presented by Harte et al. used to represent the chords in my dataset

Tag	Represents...	Children / Value
`<part>`	the chord progression	list of `<measure>`
`<measure>`	one bar	`<key>`, list of `<harmony>`
`<key>`	the key at the current bar	`<mode>`, `<fifth>`
`<mode>`	major or minor key	`"major" | "minor"`
`<fifth>`	the position on the circle of fifths relative to the middle (C or Am)	$-6 \leq integer \leq 6$
`<harmony>`	one chord	`<root>`, `<kind>`, list of `<degree>`
`<root>`	the root note of the chord	`<root-step>`, `<root-alter>`
`<root-step>`	the letter of the root note name	`"A"..."F"`
`<root-alter>`	-1 for ♭, 0 for ♮, 1 for ♯	`-1 | 0 | 1`
`<kind>`	the other notes in the chord	`"" | "m" | "7" | "m7" | "sus4" |...`
`<degree>`	alteration to a note in the chord	`<degree-value>`, `<degree-alter>`, `<degree-type>`
`<degree-value>`	interval eg. 3 = third, 5 = fifth	$0 \leq integer \leq 7$
`<degree-alter>`	-1 for minor, 1 for major interval	`-1 | 0 | 1`
`<degree-type>`	how to alter the note. (`"alter"` means make the `<degree-value>` major or minor depending on `<degree-alter>`)	`"add" | "subtract" | "alter"`

Table 4.1: iReal Pro's MusicXML Structure for a Chord Progression

4.4 Statistics

Basic stats:

- 1350 songs, 40 chords per song on average
- Most common chord types: **7** (28%), **m7** (25%), **maj7** (13%), **m7♭5** (6%), **6** (5%), **7♭9** (4%), **major** (3%), **minor** (2%)...
- Most common chord roots: **I** (22%), **V** (16%), **II** (15%), **VI** (10%), **IV** (9%), **III** (8%)...
- Most common artists: Thelonious Monk, Richard Rodgers, Cole Porter, Charlie Parker, Wayne Shorter, George Gershwin, Irving Berlin, Horace Silver, Duke Ellington, John Coltrane.

N-grams: (Only counting chord roots)

- Most common bi-grams: [V-I], [II-V], [I-I], [VI-II], [III-VI].
- Most common tri-grams: [II-V-I], [VI-II-V], [III-VI-II], [I-I-I].
- Most common 4-grams: [VI-II-V-I], [III-VI-II-V], [II-V-I-I].

As you can see from the N-grams, a clear pattern emerges from the most common sequences of chord roots. The sequence III-VI-II-V-I is a sequence of descending fifths (or ascending fourths), and is very characteristic of Jazz. This shows that the dataset is what we expected it to be, at least in this regard.

4.5 Inter-representational Parsing

I created two parsers to convert between an internal representation such as **2:[3,7,10]** and a more legible representation such as **IIm7** or **Dm7** for the purpose of debugging. The internal representation is as discussed in section 4.3, and the legible representation is as follows:

`<root><quality><extension><alteratinos><sus>`

Where:

- `<root>` is the note name or roman numeral e.g. ”D” or ”II”.
- `<quality>` is minor, major, diminished, or augmented.
- `<extension>` is additional notes like ”7”, ”maj7”, or ”9” etc.
- `<alterations>` are any flattening or sharpening of notes like ”#5” or adding new notes like ”add9”.
- `<sus>` modification or removal of the third, such as ”sus2”, ”sus4”, or ”sus”.

Chapter 5

Experimentation

5.1 Experiment Outline

In total I have six chord representations to compare, and three models to apply these representations to. I begin by explaining exactly how each of the representations work and what their main advantages and disadvantages are. Then I introduce the three models as well as a baseline model, and finally the specific experiment set-ups and model parameters.

As discussed in section 3.3, I propose to use an evaluation metric based on similarity using word embeddings alongside the accuracy-based metric. For each experiment I report the performance of each of the six representations evaluated with both the accuracy-based and similarity-based metrics. This will highlight any differences shown by the two metrics across all models and representations.

5.2 Chord Representations

5.2.1 Overview

In total, I explore six types of representations:

- One-Hot Encoding (1HE)
- Explicit Note Encoding (ENE)
- Wrapped Note Encoding (WNE)
- Normalised Note Encoding (NNE) (Contribution)
- Two-Hot Encoding (2HE) (Contribution)
- Chord embedding with Word2Vec (W2V)

These can be split into two categories: those that are content-aware, and those that are not. Content-aware chord representations are representations that encode the chord in terms of its notes in some way. ENE, WNE, NNE, and 2HE are examples of content-aware representation, while 1HE and W2V are not.

Uses of 1HE, ENE, WNE, and W2V can be seen throughout the literature, but NNE and 2HE are my proposed contributions.

Encoding chords in a content-aware way exposes the network to the notes that make up the chord, which opens up the possibility of drawing connections between chords in a way that is not possible with one-hot encoding. For example, C^7 (C,E,G,B♭) and C^9 (C,E,G,B♭,D) are functionally very similar chords (meaning they could be easily substituted without changing the meaning of the progression), which may be inferred by looking at the notes for each.

5.2.2 One-Hot Encoding (1HE)

The encoded vector has a position for each unique word in the dataset, and all values are 0 except the represented word, which is 1. Although one-hot encoding has no information about the contents of the chords, it is by far the simplest representation, which could make training easier. It is also by far the most commonly used encoding in the literature [4, 5, 6, 19].

5.2.3 Explicit Note Encoding (ENE)

The papers that do not use the one-hot encoding usually use either use this encoding (ENE) or WNE (section 5.2.4) [9, 25].

With ENE, the vector is laid out as like a keyboard, each position corresponding to a pitch in ascending order from left to right. The pitch of first note is normalised to the key of the song. The components of a chord can span up to 12 semitones, and the root of the chord can again span 12 notes, so the encoded vector is of size 24. A C major chord in the key of B would be encoded as seen in Figure 5.1, because relative to B, we have the notes B, C, C♯, D, D♯, E, F etc. and the notes in C major are C, E, G, so those are the notes that are set as 1 in the vector.

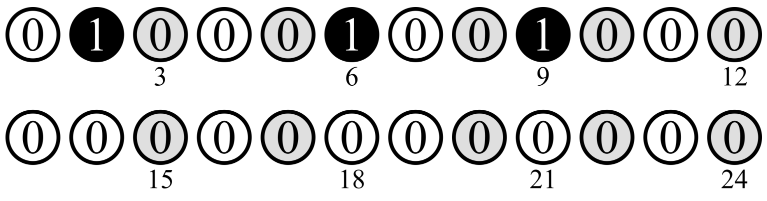

Figure 5.1: Explicit Note Encoding for C major in the key of B

This biggest issue with this encoding is this: **just because two chords have a lot of notes in common does not mean that they are similar chords.** The chords C^{maj7} and Dm (C,E,G,B vs E,G,B) have very similar notes, the only difference being an extra C in the C^{maj7}. This encoding would make these two chords very similar, although in reality the chord progression would be changed significantly if a Dm was substituted with a C^{maj7}.

5.2.4 Wrapped Note Encoding (WNE)

WNE is the same as ENE, except that notes after the 12th position in the vector are transposed an octave down, so that the vector is only of size 12. For example, in ENE, a G^7 chord (G,B,D,F) in the key of C would have a 1 in positions 7, 11, 14, and 17, but in WNE it would be in positions 7, 11, 2, and 5 (Figure 5.2).

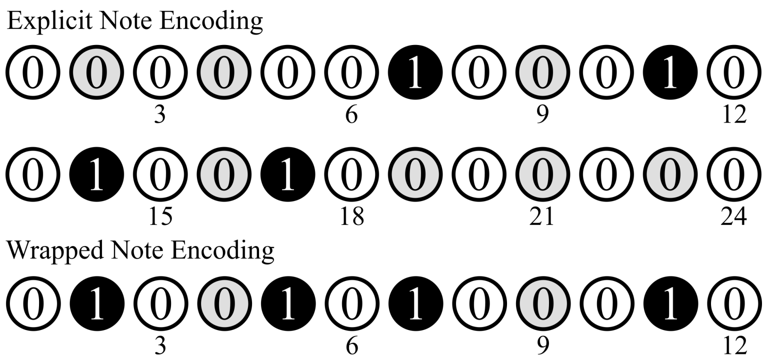

Figure 5.2: Explicit and Wrapped Note Encoding for G^7 in the key of C

This encoding suffers from the same issue that explicit note encoding has, but it is likely to perform better because the input is simplified, since the same note can no longer be in two possible positions.

5.2.5 Normalised Note Encoding (NNE)

I propose to build on ENE with NNE, to introduce more structure and information into the representation. The chord is split into two parts: the root note, and the rest. The first 12 positions in the vector is for one-hot encoding the root note, which is one of 12 possible notes in an octave. The other 11 positions encode the other notes that are in the chord (I assume the root note to always be present in the chord, so we only need 11 positions).

A G^7 (G,B,D,F) in the key of C would be encoded as seen in Figure 5.3, because G is 7 semitones above C, B is 4 above G, D is 7 above G, and F is 10 above G.

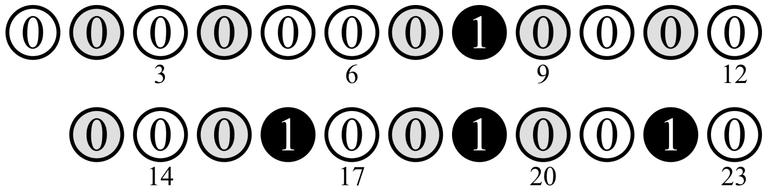

Figure 5.3: Normalised Note Encoding for G^7 in the key of C

This encoding solves the main issue with ENE because it not only captures what notes are in the chord, but also how they are related to each other. For example, all seventh chords are identical apart from the first position.

Taking the same example from section 5.2.3, comparing C^{maj7} with Dm in the key of C would be encoded as seen in Figure 5.4, which is not similar at all, just what we want.

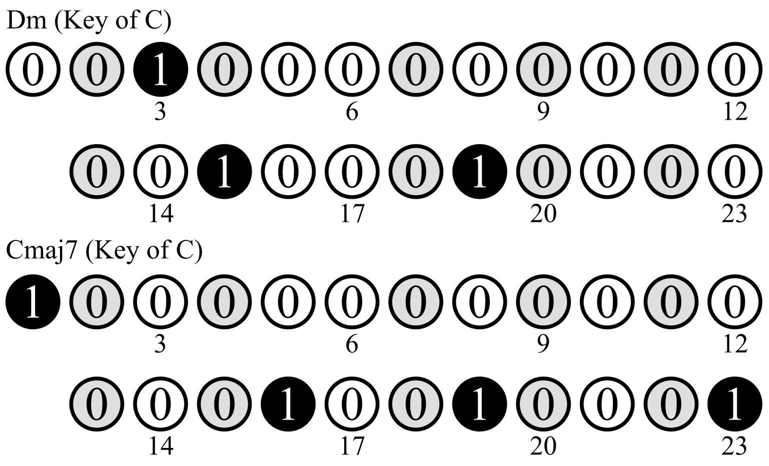

Figure 5.4: Normalised Note Encoding for Dm and C^{maj7} in the key of C

5.2.6 Two-Hot Encoding (2HE)

As an alternative to NNE, I propose 2HE which is a middle-ground between 1HE and NNE. 2HE is also split into two parts, but here the second part is just a one-hot vector of all the different chord component types. The size of this vector will depend on how many different types of components exist in the dataset. Some examples of chords in the key of C can be seen in Figure 5.5.

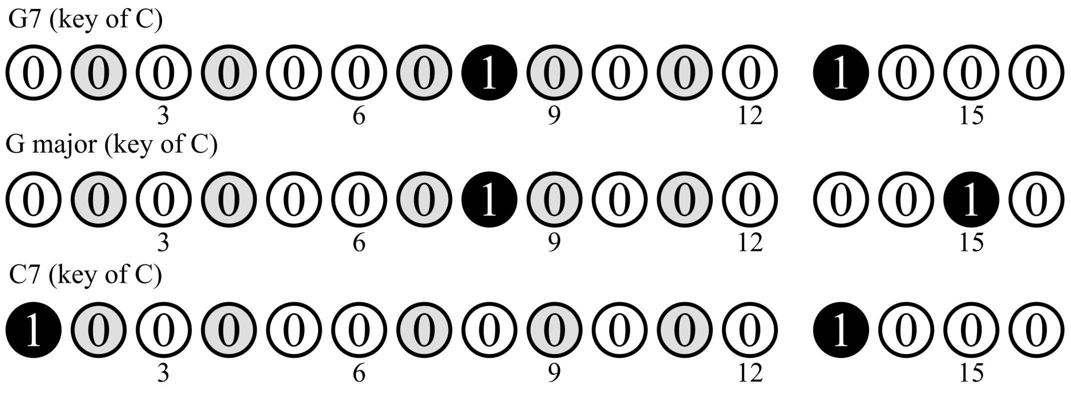

Figure 5.5: Two-Hot Encodings for G^7, G major, and C^7 in the key of C with 4 different chord component types

5.2.7 Chord Embeddings (W2V)

Chord Embeddings are machine-learned representations of chords where chords with similar meaning have similar vectors. I use Gensim's [30] Word2Vec library to do this, and I explore the following parameters to best fit my dataset:

- **Vector size:** the size of the vector created by Word2Vec. Too small and the model could struggle to retain enough information in the embedding; too large and it may be too complex a representation to learn from in the downstream task.

- **Context window size:** the number of chords that are considered around the target chord. Too few and the model will have too narrow a "vision"; too many and it may be difficult to train the model.
- **Number of iterations:** how many iterations the model learns for. Too few and it will not be able to learn; too many and the model will stop being good at generalising and overfit.

Because learning chord embeddings is an unsupervised task, I will need to evaluate my Word2Vec model using a downstream task [31]. I use a Feed-Forward Neural Network with an accuracy-based evaluation for this downstream task, with model parameters that worked best for 1HE (section 5.3.2).

The Chord2Vec paper [20] explores two other chord embedding models based on the Neural Autoregressive Distribution Eliminator [17] and the Sequence-to-Sequence model [33], which would make excellent areas to explore in my work, however, I will leave this for future research because the Gensim library does not support these models. Gensim's Word2Vec model also only takes one-hot vectors as input, so I will also leave using the content-aware chord representations (section 5.2.1) as the inputs for future research.

5.3 Models

5.3.1 N-gram Baseline

N-grams are a good non-neural network based model to use as a baseline, as they have been used many times in the past for music generation [1, 28, 41]. N-gram models work as follows:

1. Take only the last $N-1$ chords from the input progression.
2. Find all examples of this exact progression in the dataset.
3. Look at the chords that appear after each of these examples.
4. Find the most common chord, and report this as the final answer.

The problem with this is that if the progression has never been seen before, it cannot suggest anything. To combat this, I have explored the use of Katz's back-off model [15] which decreases N until there exist enough examples in the dataset (above a threshold k), so that the model can suggest a result with a certain amount of confidence (depending on k). Setting k to 1 means if the there is just one example of the input progression in the dataset, the chord that follows that example will be the final answer. If k is set to 2, there will need to be at least two examples, and so on.

Note that if for example, k is set to 2, that does not mean that the model necessarily has only two examples to use to find the most popular chord, it just means that if there are less than two examples, N is decreased by 1 in the hopes of finding more than two examples.

I report the performance of my n-gram models as both accuracy and similarity. Both of which will be based on the validation set, which is not used to create the n-grams.

5.3.2 Feed-Forward Neural Network (FNN)

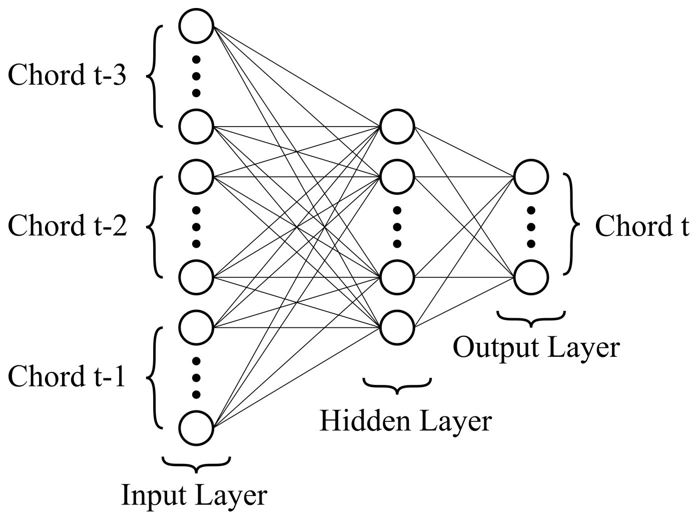

Figure 5.6: FNN model architecture with 3 input chords and 1 hidden layer

The Fully-Connected Feed-Forward Neural Network (FNN) is the simplest neural network based model, which makes it a good first model to try.

The network's input layer is a concatenation of the input chords in their corresponding vector representations (Figure 5.6). Note that the length of the vectors depends on the representation used. For example, if the representation used is ENE (section 5.2.3), then the vectors for each chord is of length 24, and if there are 3 input chords, the length of the whole input vector is $3 \times 24 = 72$.

Just like the number of input chords, the number of hidden layers and the number of hidden units are also variable. These parameters are part of my parameter search for the FNN experiments.

During training, the target label for the output chord t is always a one-hot encoding of the real chord. During testing, the unit with the highest value is selected as the final answer for the model.

5.3.3 Convolutional Neural Network (CNN)

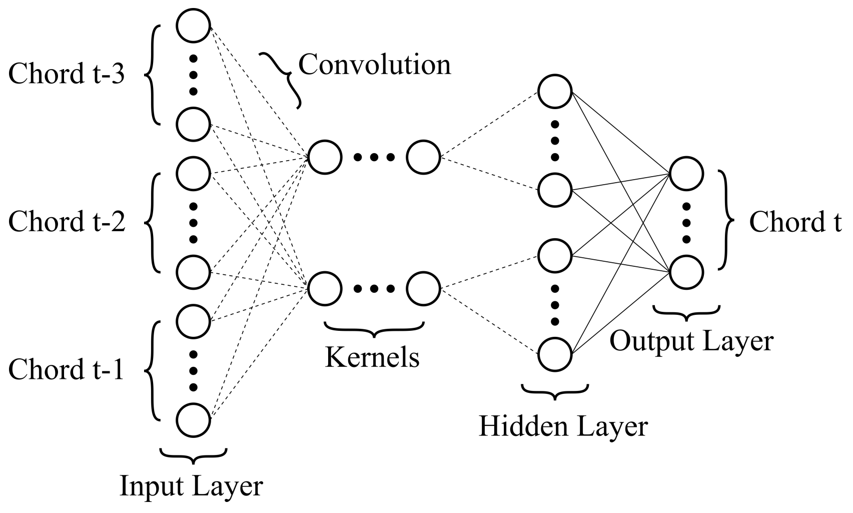

Figure 5.7: CNN model architecture with 3 input chords, 2 kernels, and 1 hidden layer

The advantage of CNNs comes from the fact that some weights between nodes are shared (Figure 5.7). The weight sharing is set up in such a way to allow exploiting the local relationships between nodes in the input layer. **This might be particularly effective with the content-aware representation, where the notes of the chords are exposed to the network (section 5.2.1).** For example, with NNE the network might be able to pick up and learn from patterns such as Major, Major, Minor, etc. because the chord components are normalised to the root.

If we imagine the input progression, not as a 1D concatenation of chord vectors, but as a 2D matrix of size $m \times n$ where m is the number of inputs and n is the number of channels, then a 1-dimensional convolution can exploit the local relationships or the order of inputs in the m direction, and not in the n direction. When we convert our chord representations to this matrix we have two options as to what we consider to be the inputs and channels (see Figure 5.8):

- **Each input is a chord and the channels for that input is the chord representation vector**, so $m =$ the context window size, and $n =$ the chord representation size. This means that *temporal* information can be exploited, since the weights for each note are shared across different chords in time, and potentially putting an emphasis on the absolute positions of the notes.
- **Each input is a position in the chord representation vector**, and the channels for that input are all the chords in the progression, so $m =$ the chord representation size, and $n =$ the context window size. This means that *pitch* information can be exploited, since it has a more localised view of the notes within a single chord, potentially putting more emphasis on the relative positions of the notes. We can achieve this method very simply by transposing the input matrix.

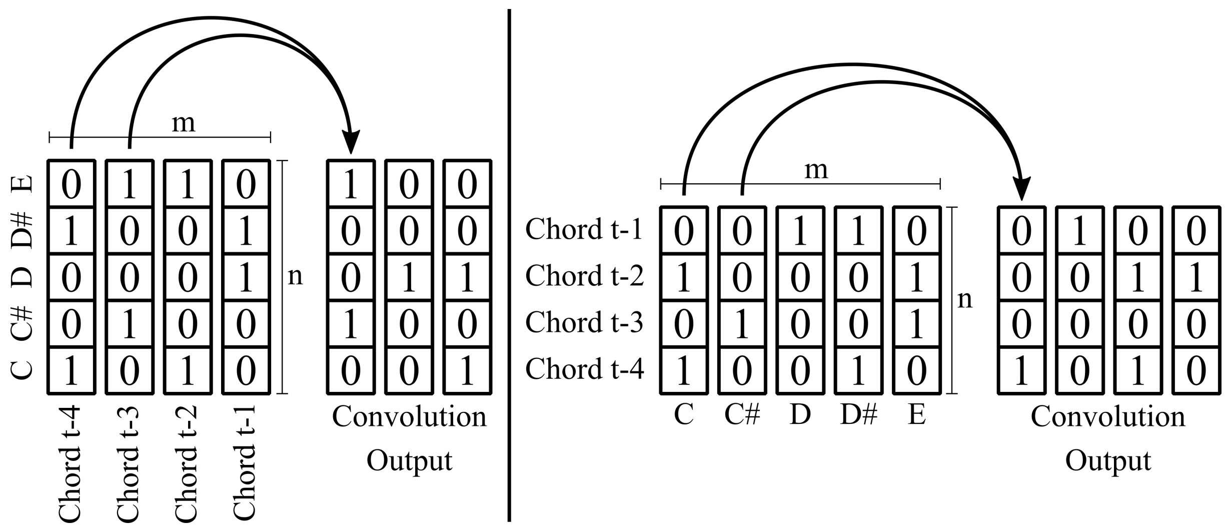

Figure 5.8: CNN input for temporal locality (left) and CNN input for pitch locality (right), both with 4 input chords, and a chord representation vector length of 5

5.3.4 Long Short-Term Memory Network (LSTM)

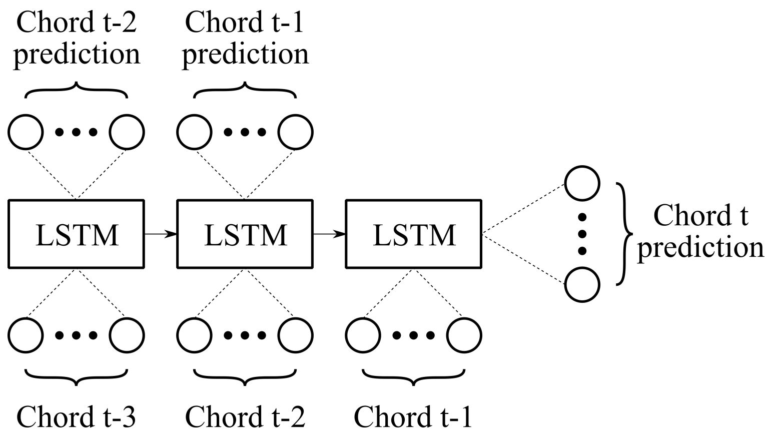

Figure 5.9: LSTM model architecture with 3 input chords

LSTMs are the current state of the art model for chord generation, so I expect it to outperform the other models. **Each chord in the input progression is fed into the network sequentially one by one (Figure 5.9), and after each chord, the LSTM is able to make prediction for the next chord.**

LSTMs expand on RNNs by including a *memory cell* and three gates: *forget, input,* and *output*. These gates manipulate the output of the LSTM before it is fed back into itself, and overall it should be able to handle longer term dependencies better than a standard RNN.

It has been shown that one of the most important parameter when setting up an LSTM is the forget gate bias [35]. The values in the forget gate are learned during training, and they govern how quickly older inputs are forgotten. The values of the forget gate are put through a sigmoid function to be squashed between 0 and 1, and are then multiplied with the previous LSTM's output. So, rather counter intuitively, the higher the values in the forget gate, the more it remembers and the less it forgets. The bias for the forget gate is a fixed number added to all the values in the forget gate.

Gers et. al. [10] first proposed the idea that the forget gate bias should be set to a higher value like 1 or 2, which was also reemphasised by Jozefowicz et. al. [14]. The idea is that if the forget gate is too low at the beginning of training, it would introduce a vanishing gradient which could hinder learning the longer-term dependencies. The solution is to give the bias a higher value, so that at the beginning of the training process, the LSTM can remember more, allowing gradient flow. It can then reduce how much it remembers over the course of training to optimise performance.

5.4 Implementation Details

Train / Validation sets: I use a 85/15 split for training/validation sets.

Word2Vec model: I use Gensim's [30] Word2Vec model for this task, which is a Skip-Gram Feed-Forward Neural Network. For the downstream task, I use an FNN with: Context window = 4; hidden layers = 1; hidden units: 500; Dropout, Batch Norm, and ReLu in that order after the hidden layer; Adam optimisation [16]; Batch size = 100; learning rate = 0.0005.

N-Gram model: The N-Gram model is implemented using algorithm 1, and starts with N = 10. k will be explored with values 1-32.

Algorithm 1 Predict Chord with N-Grams and back-off

```
Input: input chords $c_i$, dataset $d$, threshold $k$
Initialise $prediction = null$
Initialise $contextWindow = c.length$
while $prediction == null$ or $contextWindow < 1$ do
    Initialise $nGrams = buildNGrams(n = contextWindow, data = d)$
    Initialise $inputSubset = lastNOf(array = c, n = contextWindow)$
    Initialise $probabilities = nGrams.find(inputSubset)$
    if $probabilities.length >= k$ then
        $prediction = probabilities.getMostLikely()$
    else
        $contextWindow = contextWindow - 1$
    end if
end while
return $prediction$
```

FNN: All neural networks are implemented using PyTorch [27]. The FNN uses: Dropout, Batch Norm, and ReLu in that order after each hidden layer; Adam optimisation [16]; Batch size = 100; learning rate = 0.0005; hidden layers = 1 - 3; context window = 1 - 11.

CNN: The CNN uses: Dropout, Batch Norm, and ReLu in that order after each hidden layer; Adam optimisation [16]; Batch size = 100; Learning rate = 0.0005; Hidden layers = best result from FNN; Context window = best result for FNN; Convolution kernels = 1 - 4 for normal CNN, 1 - 12 for transposed CNN.

LSTM: The LSTM uses: Dropout, Batch Norm, and ReLu in that order after the LSTM; Adam optimisation [16]; Batch size = 100; Learning rate = 0.0005; Context window = best result for FNN; Forget gate bias = -2 - 4.

Early Stopping for Neural Networks: For all the experiments involving a neural network, I continue training until the validation accuracy no longer goes up. Training ends when it has been longer than 100 episodes since the accuracy was last increased.

Chapter 6

Results

In this chapter, I present the results for the experiments that I ran according to the implementation details in section 5.4. I interpret the results based on the theory discussed in chapter 5, and provide possible explanations if the results did not match what I expected.

In all my experiments, I refer to *accuracy* as the total number of correct predictions divided by the total number of predictions in an unseen validation set. Each model's final answer is taken as the node in the output layer with the highest value, where each node corresponds to a unique chord in the dataset.

I refer to *similarity* as the mean cosine similarity between the Word2Vec vector representations of the expected and predicted chords in an unseen validation set. Again, each model's final answer is taken as the node in the output layer with the highest value, which is then converted to a Word2Vec vector.

6.1 Summary

Table 6.1 shows the results for the various baseline models, and table 6.2 shows the performance of each representation for each of the best performing models with both accuracy and similarity based evaluation. Specifically, I found the following model parameters to perform best:

- **N-gram Model:** $k = 8$.
- **W2V:** 100 vector size, 100 iterations, 4 context window.
- **FNN:** 1 hidden layer, 10,000 hidden units, 4 context window.
- **CNN:** kernel size of 3 without transposition, 1 hidden layer, 2,000 hidden units, 4 context window.
- **LSTM:** 1.0 forget gate bias, 4 context window.

Baseline	Accuracy	Similarity
Random Guess	0.22%	0.06
N-gram Model	37.47%	0.5564
Descending Fifths	**41.27%**	**0.5851**

Table 6.1: Baseline performances.

Model	Representation	Accuracy	Similarity
FNN	**1HE**	**44.08% ± 1.47**	**0.5980 ± 0.0136**
	W2V	43.58% ± 1.41	0.5947 ± 0.0187
	2HE*	43.43% ± 0.89	0.5921 ± 0.0089
	NNE*	42.73% ± 1.50	0.5863 ± 0.0097
	ENE	42.70% ± 0.79	0.5890 ± 0.0098
	WNE	41.99% ± 1.46	0.5765 ± 0.0167
CNN	**1HE**	**45.62% ± 1.37**	**0.6145 ± 0.0175**
	W2V	44.30% ± 1.53	0.6046 ± 0.0081
	2HE*	45.12% ± 1.44	0.6022 ± 0.0147
	NNE*	44.91% ± 0.93	0.6025 ± 0.0155
	ENE	43.82% ± 1.03	0.6017 ± 0.0095
	WNE	41.59% ± 1.15	0.5799 ± 0.0149
LSTM	**1HE**	**52.73% ± 1.18**	**0.6462 ± 0.0151**
	W2V	49.13% ± 1.33	0.6330 ± 0.0176
	2HE*	47.65% ± 1.66	0.6101 ± 0.0065
	NNE*	46.80% ± 0.91	0.6094 ± 0.0061
	ENE	45.79% ± 1.60	0.6005 ± 0.0184
	WNE	43.66% ± 1.45	0.5924 ± 0.0093

Table 6.2: Accuracy and similarity performance for all representations and for each all models, with standard deviation across 5 runs. Each model's parameters are based on what I found to give the highest performance. * proposed representations.

6.2 Word2Vec

6.2.1 Parameter Exploration

The first experiment is an exploration of the effects of various vector sizes for the downstream FNN task, shown in figure 6.1. **As the vector size increases, the accuracy increases significantly up until a size of around 100, where it levels out at around 42%.** The training time also increases with larger vectors, so I have chosen a vector size of 100 for the final model.

The second experiment is an exploration of the effects of various window sizes for the downstream FNN task, shown in figure 6.2. Nnote that this window size is just for the chord embedding task, which is not the same as the context window size for the downstream FNN task. **Curiously, the figure shows no significant trend in the change of window sizes, when I expected the accuracy to be lower for very small and very big values.** Upon further reading, I found that Levy et. al. [18] say that large windows tend to capture the relationships between words more, small windows tend to capture more about the word itself, and that both can be effective depending on the situation. It is possible that for the purposes of chord embedding, relationship information and individual chord information is equally important. I have chosen a window size of 4 for the final model, just to match the context window size for the chord generation task.

The final experiment is an exploration of the number of iterations to train the Word2Vec model, with the accuracy for the downstream FNN task, show in figure 6.3. **The figure shows no significant improvements to accuracy past 100 iterations, while taking longer to train, so I have chosen 100 for the final model.**

The final Word2Vec model has the following parameters: size 100 vector, 100 iterations, size 4 window.

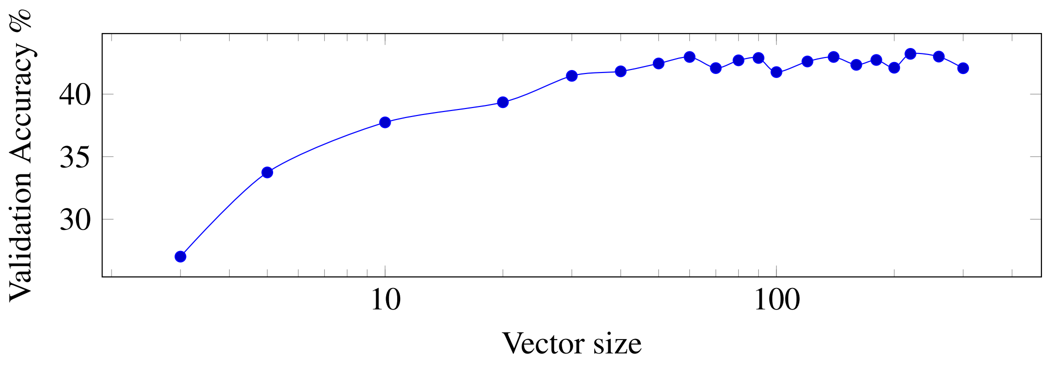

Figure 6.1: Validation accuracy for Word2Vec vector sizes (log scale) with window size 4, 100 iterations, and a FNN downstream task (1 hidden layer, 500 hidden units, 100 batch size, 0.0005 learning rate, Adam optimisation, Dropout, Batch Norm, and ReLu) showing that large vector sizes increase accuracy, but only up to around 100. Highest value: 43.22% at 220.

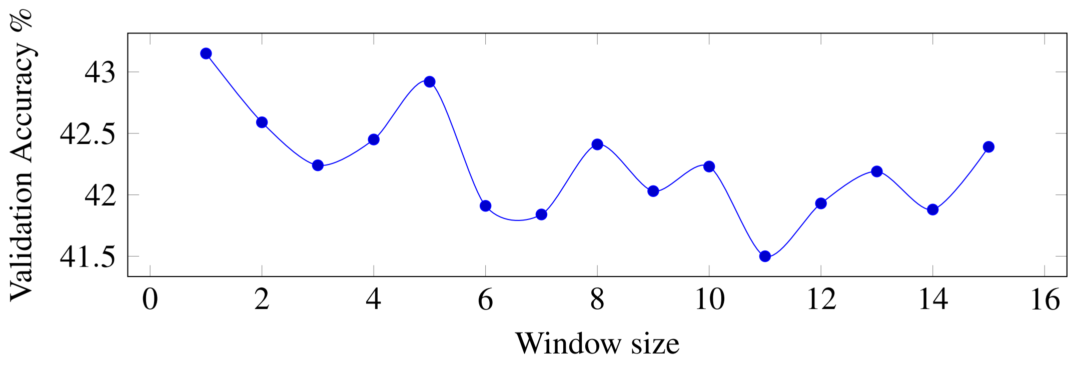

Figure 6.2: Validation accuracy for Word2Vec window sizes with 100 iterations, vector size 100, and a FNN downstream task (1 hidden layer, 500 hidden units, 100 batch size, 0.0005 learning rate, Adam optimisation, Dropout, Batch Norm, and ReLu) showing no significant trend in the change of window size. Highest value: 43.15% at 1.

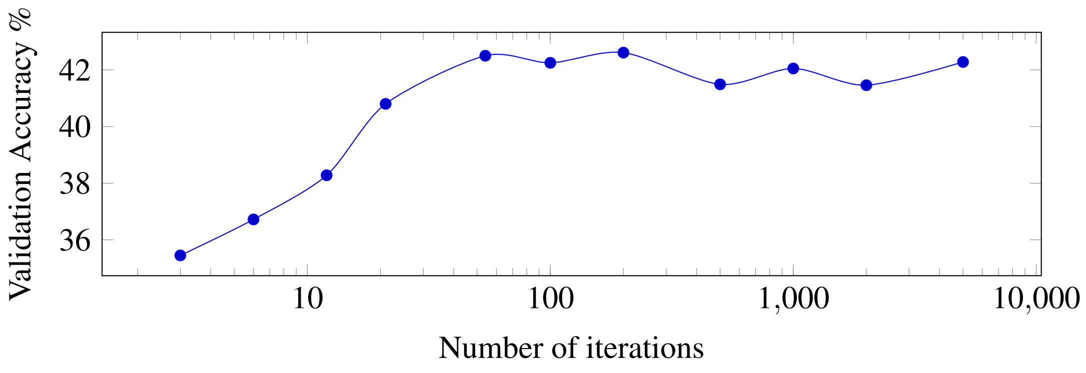

Figure 6.3: Validation accuracy for Word2Vec trained for different number of iterations (log scale) with window size 4, vector size 100, and a FNN downstream task (1 hidden layer, 500 hidden units, 100 batch size, 0.0005 learning rate, Adam optimisation, Dropout, Batch Norm, and ReLu) showing that more iterations yields better accuracy, but only up to around 100. Highest value: 42.61 at 200.

6.2.2 Subjective Insights into Word2Vec Results with PCA

I end my Word2Vec parameter exploration with some subjective insights into the chord embedding I have created. Figure 6.4 shows the Principle Component Analysis (PCA) for each encoded chord using the final Word2Vec model. The chords are represented with numbers that correspond to the number of semitones that the chord root is above the key (0-11), followed by the chord type. For the purpose of readability, I have reduced the number of unique chords by applying simplification level 1 (section 5.3.1) to generate this figure.

The first thing to note about the figure is that **two chords whose roots make a perfect fifth have also been encoded with similar vectors**, this has been highlighted by the black lines between pairs of chords. Because there are 12 semitones in an octave, a fifth, which is 7 semitones, can be recognised by observing a difference of 7 or 5 between chords. Almost every chord has a neighbour that is a perfect fifth away.

In section 4.4, I calculated the most common n-grams and found that by far the most common are chains of descending fifths. This is to be expected, since it is very characteristic of jazz, and can indeed be seen very clearly as chains in the PCA plot. Note that like the n-gram model, the Word2Vec model does not have access to any information about the chord itself, so the similarity between fifths in the encoded vector has come entirely from context.

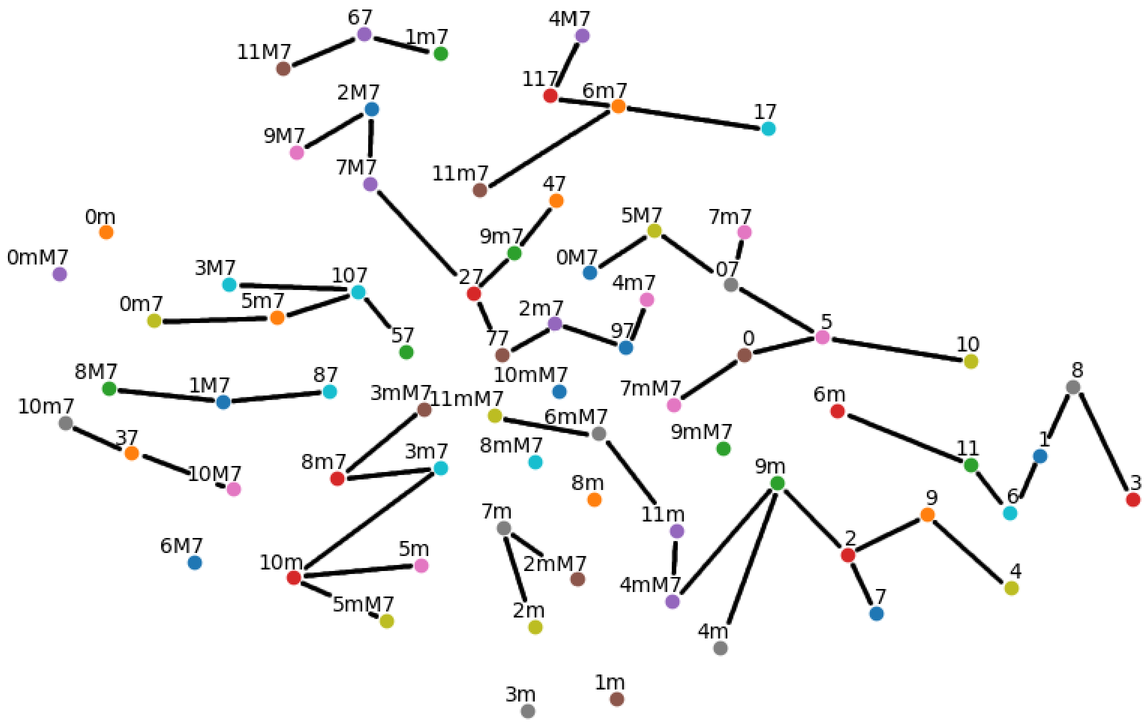

Figure 6.4: PCA of the dataset encoded with the final Word2Vec model (vector size: 100, window size: 4, iterations: 100) with black lines connecting two similar chords that also make a perfect fifth (+7 or -5 semitones). The chord labels are the semitones above the key + the chord type.

6.2.3 Subjective Insights into Word2Vec Results with Word Analogy

Finally, I provide more insight into the created Word2Vec model by looking at word analogies. The original Word2Vec paper by Mikolov et. al. [23], showed that the created vectors can be combined to create a new vector that is similar to what you would expect logically. A famous example is $King - Man + Woman = Queen$, which works because the difference in the vectors for $King$ and $Queen$ is very similar to the difference in the vectors for Man and $Woman$, or in other words $Queen - King = Woman - Man$, which is just a rearrangement of the example equation.

I show the quality of my Word2Vec embedding by creating the chains of descending fifths using word analogy. For example, I show that a II chord is to a V chord in the same way that a V chord is to a I chord, i.e. a fifth. As an equation this would take the form $V - I = II - V$ or $V - I + V = II$, which can be seen in table 6.3.

Combination	Most Similar Chords (Cosine Similarity)			
V^{7} - I + V^{7}	**IIm^{7} (0.62)**	I^{maj7} (0.55)	VII^{7} (0.54)	IIIm^{7} (0.53)
V^{maj7} - I^{7} + IIm^{maj7}	**VI^{7} (0.51)**	V^{maj9} (0.43)	IIIm$^{\flat 13}$ (0.41)	$\flat$VI^{69} (0.41)
V^{7} - I^{7} + VI^{7}	**IIIm^{7} (0.55)**	IIm^{7} (0.53)	I^{maj7} (0.46)	VIm^{9} (0.41)

Table 6.3: Word Analogies for final Word2Vec model.

6.3 N-gram Baseline

In the search of finding the best performing n-gram model, I ran multiple experiments with different values of k. To recap, k is the threshold for the number of matched examples in the dataset, below which the n-gram model will shorten the input chord sequence. I discuss this in more detail in section 5.3.1.

Low values of k means that a longer sequence is more likely to matched in the dataset, at the cost of having fewer examples to calculate probabilities with. High values of k, means that the model will sacrifice the length of the input sequence to find more examples in the dataset.

Table 6.4 shows the performance for each value of k both with accuracy and similarity, with an additional row for the input progression length N fixed to 2. I set N to 10, to give the model as much headroom as it could need, but even at $k = 1$, it would almost always be shortened to around 4.

N	k	Accuracy	Similarity	Mean N	Mean P
10	1	34.05%	0.5237	3.8	16.9
10	2	34.45%	0.5298	3.7	16.5
10	4	35.77%	0.5352	3.4	23.2
10	**8**	**37.47%**	**0.5564**	**3.0**	**33.8**
10	16	35.33%	0.5372	2.7	48.1
10	32	33.39%	0.5192	2.4	70.3
2	1	32.79%	0.5103	2.0	75.7

Table 6.4: Performance and statistics of the n-gram model for various values of k with an additional row for N fixed to 1. Mean N is the mean number of chords used to give the final answer. Mean P is the mean number of examples found when the final answer was given.

I also explored the use of chord simplifications as an alternative to Katz's back-off model (discussed in section 5.3.1). I achieved this by modifying the notes in the chords to remove some complexity, leaving only the essence of each chord. More specifically, I take only the simple triads for each chord.

However, I realised that this particular exploration was not fruitful. First of all, I found the simplification to yield very bad results when evaluated with the non simplified dataset. This is mainly because the model can only generate simplified chords, which are actually not common in the dataset at all. In the dataset, only 5% of chords are simple triads, so the best it could ever achieve in terms of accuracy is 5%. Secondly, this model would be quite limited in a practical scenario since there are many chords it would simply not be able to generate, and so chord simplification was not something I wanted to pursue any further.

6.4 Other Baselines

6.4.1 Random Guess

There are 464 unique chords in the dataset, so a random guess will be correct once in 464 times, or **0.22% accurate**. On average, a random guess will also come out with a **W2V similarity score of 0.06**.

6.4.2 Descending Fifths Rule

A chord progression with descending fifths is very characteristic of Jazz, and I have shown that my dataset captures this characteristic well. For this baseline, the generated chord is always the fifth below the last chord in the input progression.

Over the 53,000 chords in the dataset, this rule will get an **accuracy of 41.27%**, and a **similarity score of 0.5851**, which is surprisingly high given the simplicity of the rule.

6.5 Fully-Connected Neural Network (FNN)

6.5.1 Context Window Size

The first experiment I run to fine-tune the FNN, is a grid search of the size of context window for the input (Figures 6.5 and 6.6). For the rest of the network, I start with 1 hidden layer with 500 hidden units. **The figures show that a context window size of 4 works the best overall, with 1HE performing the best with an 42.84% accuracy / 0.5967 similarity.** W2V follows closely behind with 42.69% accuracy / 0.5898 similarity. Other encodings fall behind significantly and, with the exception of WNE which performed even worse, seem to perform fairly similarly to each other.

6.5.2 Hidden Layers and Units

The next experiment is an exploration in the number of hidden layers and the number of hidden units in each layer. I ran a grid search of hidden units for a single hidden layer, and the results are shown in figures 6.7 and 6.8. **Both figures clearly show that an increase in hidden units results in an increase in accuracy and similarity.** Similar to the context window experiments, 1HE leads consistently, followed closely by W2V, both with their highest scores at 2,000 or 10,000 hidden units. However, the trend continues to go up, so I believe that with even more units, more performance could be found.

Next, I ran experiments with various numbers of hidden layers and hidden units, to see if a deeper network could be beneficial. **Tables 6.5 and 6.6 show that, of the architectures I chose to run, two layers with 500 units each performed the best.** Both 1HE and W2V performed best again, with marginal difference in performance between them. Two layers seem to perform better than a single layer, but still not as good as a single wide layer.

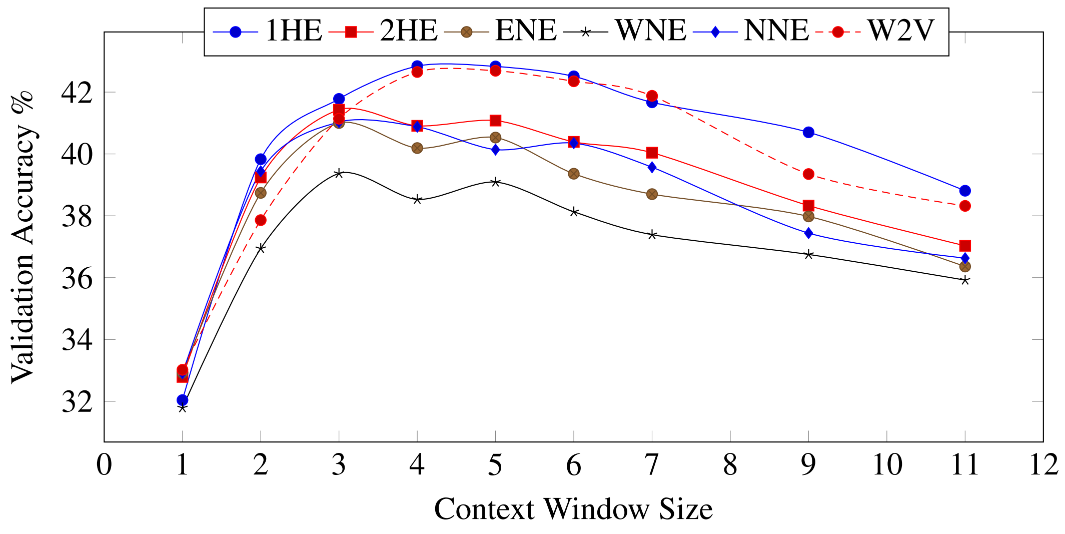

Figure 6.5: FNN with 1 hidden layer and 500 hidden units. Max values: **1HE**: 4 = 42.84%, **W2V**: 4 = 42.69%, **2HE**: 3 = 41.43%, **NNE**: 3 = 41.03%, **ENE**: 3 = 41.00%, **WNE**: 3 = 39.37%.

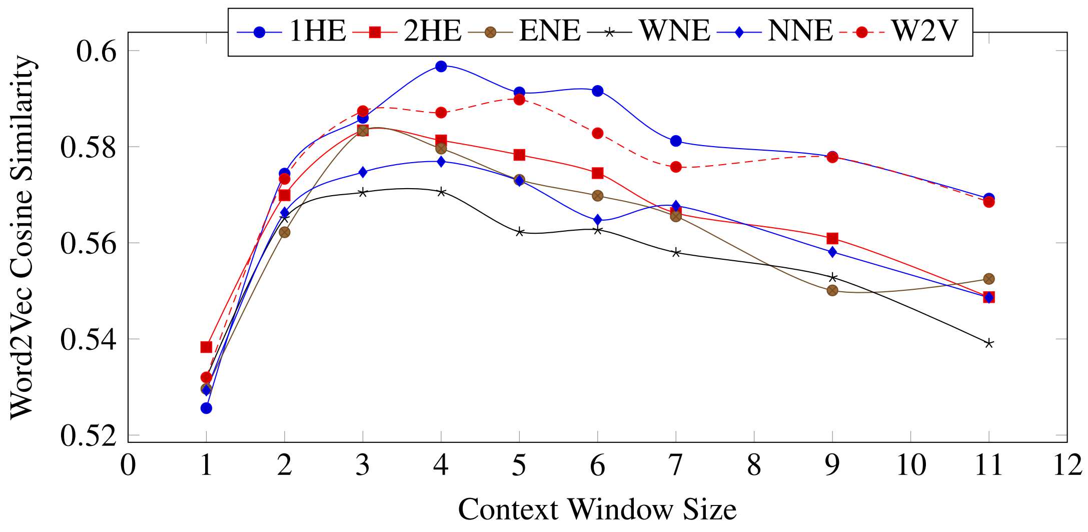

Figure 6.6: FNN with 1 hidden layer and 500 hidden units. Max values: **1HE**: 4 = 0.5967, **W2V**: 5 = 0.5898, **2HE**: 3 = 0.5834, **ENE**: 3 = 0.5833, **NNE**: 4 = 0.5769, **WNE**: 4 = 0.5706.

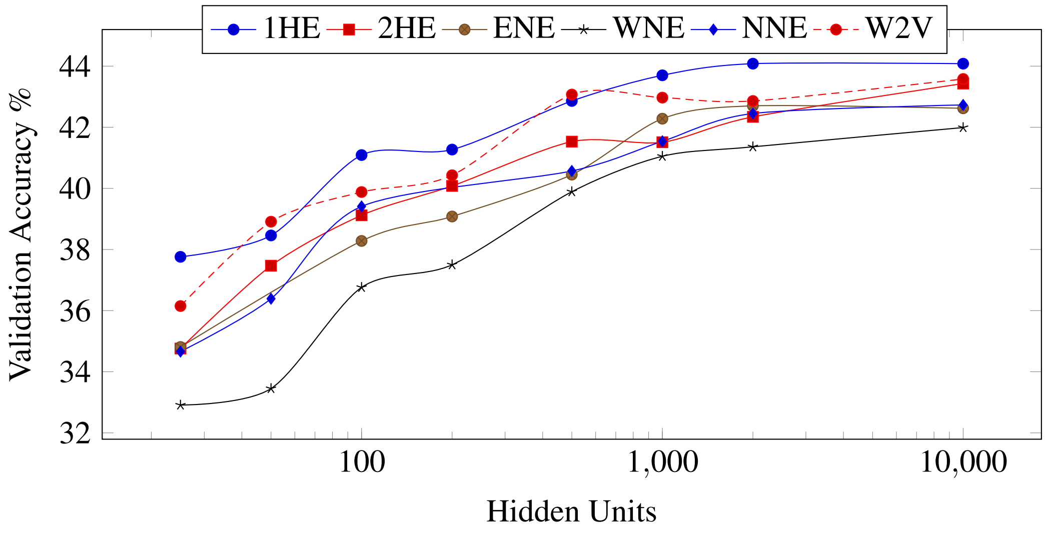

Figure 6.7: FNN with 1 hidden layer and context window 4. Max values: **1HE**: 2,000 = 44.08%, **W2V**: 10,000 = 43.58%, **ENE**: 2,000 = 42.7%, **NNE**: 10,000 = 42.73%, **2HE**: 10,000 = 43.43%, **WNE**: 10,000 = 41.99%.

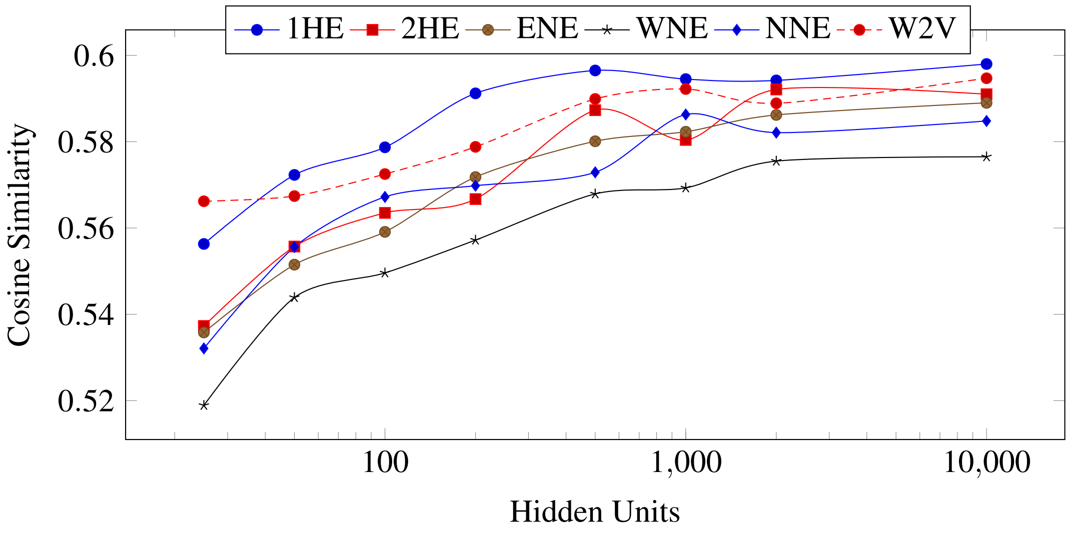

Figure 6.8: FNN with 1 hidden layer and context window 4. Max values: **1HE**: 10,000 = 0.598, **W2V**: 10,000 = 0.5947, **2HE**: 2,000 = 0.5921, **NNE**: 1,000 = 0.5863, **ENE**: 10,000 = 0.5890, **WNE**: 10,000, 0.5765.

Layers	**1HE**	W2V	2HE	NNE	ENE	WNE
500/500	**43.12%**	**42.41%**	**41.61%**	**41.69%**	**41.30%**	**39.96%**
500/100	41.68%	40.39%	40.16%	38.92%	40.26%	39.96%
500/300/100	41.62%	40.80%	40.45%	40.00%	40.01%	38.41%
300/300	41.88%	42.11%	39.58%	40.19%	40.31%	38.09%
200/200/100	39.85%	39.12%	39.47%	38.84%	38.16%	38.93%
100/100/100	38.57%	37.92%	37.72%	37.54%	38.08%	36.28%

Table 6.5: Validation accuracy for different number of layers and hidden units of FNN with context window 4. Notation: layer1/layer2/layer3...

Layers	**1HE**	W2V	2HE	NNE	ENE	WNE
500/500	**0.5936**	**0.5851**	**0.5795**	**0.5764**	**0.5721**	**0.5683**
500/100	0.5730	0.5863	0.5711	0.5691	0.5670	0.5612
500/300/100	0.5700	0.5795	0.5657	0.5704	0.5718	0.5636
300/300	0.5810	0.5898	0.5700	0.5618	0.5670	0.5616
200/200/100	0.5700	0.5785	0.5567	0.5560	0.5663	0.5561
100/100/100	0.5600	0.5617	0.5562	0.5564	0.5562	0.5474

Table 6.6: Similarity score for different number of layers and hidden units of FNN with context window 4. Notation: layer1/layer2/layer3...

6.5.3 Generalisation

Normally, one would expect very wide and shallow networks to overfit and decrease in validation accuracy as training accuracy approaches 100%. **However, in this FNN I saw that validation accuracy remained at its highest point, regardless of how many additional epochs the model was trained for (see Figure 6.9).** In section 6.2, we observed something similar; increasing the number of iterations does not cause the validation accuracy to fall.

The validation accuracy would normally fall because the model starts to memorise the data exactly, rather than coming up with a more general idea. My model does not do this, which tells us that for this particular task, memorising the data exactly is actually the optimal strategy. I discuss this in more depth in the conclusions chapter (section 7.1).

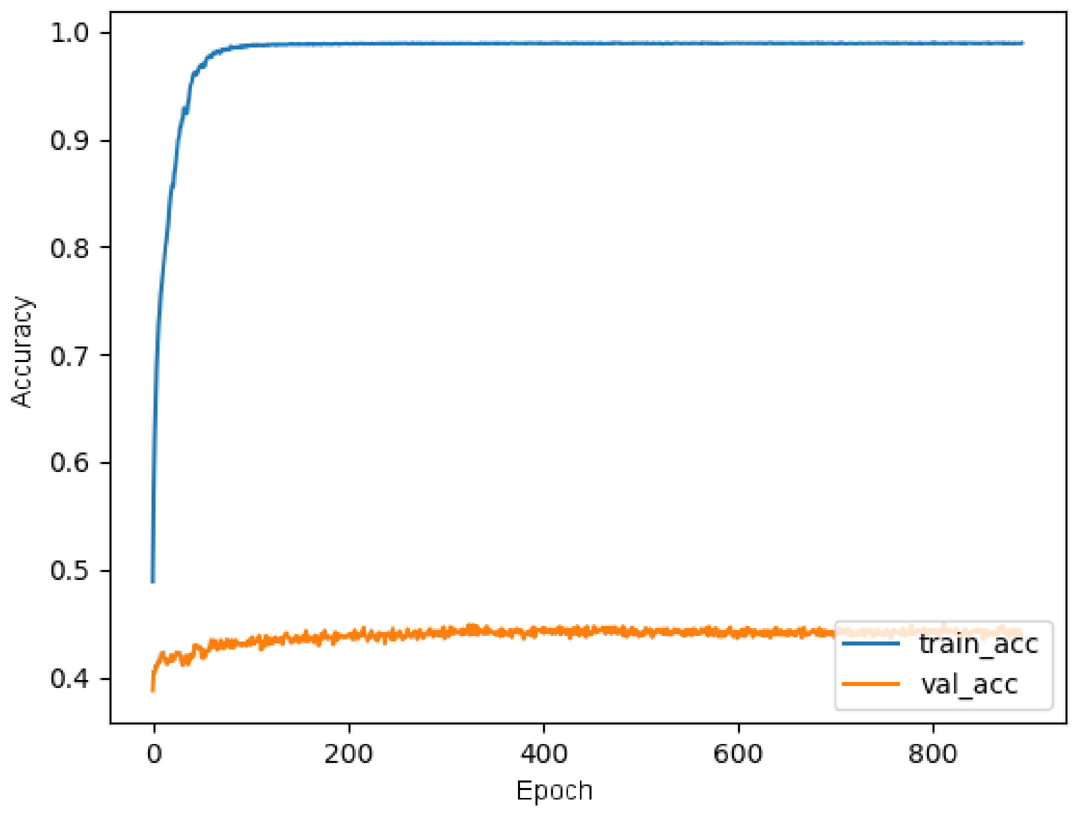

Figure 6.9: FNN training graph with 1 hidden layer, 500 hidden units, 4 context window.

6.6 Convolutional Neural Network (CNN)

6.6.1 CNN for Temporal Locality

The first experiment is the CNN that can exploit temporal locality, so without transposing of the input matrix (figures 6.10 and 6.11). The kernel size can only go up to 4, because there are only 4 input chords. While there seems to be no obvious trend for the effects of kernel size, the CNN performs does seem to perform slightly better than FNN, with the one-hot encoding getting 45.62% accuracy / 0.62 similarity over the FNN's 44.08% accuracy / 0.5942 similarity with similar settings.

6.6.2 CNN for Pitch Locality (Transposed)

The second experiment is the CNN that can exploit pitch locality, so with transposing of the input matrix (figures 6.12 and 6.13). Because the input matrices are transposed, we can increase the kernel sizes up to 12, which is the length of the smallest chord representation, WNE. Again, there is no obvious trend to be seen, and this CNN performs very similarly to the temporally-exploiting CNN. The one-hot encoding achieves a maximum of 45.7% accuracy / 0.6208 similarity.

6.6.3 Comparison

In section 5.3.3, I discussed how I expected the first (not transposed) CNN to favour chord representations with absolute note positions, and the second CNN to favour chord representations with relative note positions. 2HE and NNE are examples of the former, because the notes of the chord are normalised relative to the chord root, and ENE and WNE are examples of the latter.

Looking at the performance of 2HE and NNE in figures 6.10, 6.11, 6.12, and 6.13, reveals a very subtle increase in accuracy (1-2%) for the first (not transposed) CNN compared to the transposed CNN, however, the same cannot be said for similarity or even for ENE and WNE in the second CNN. Unfortunately the results are a little too noisy to be able to clearly see this effect, perhaps with more data this would be easier to see.

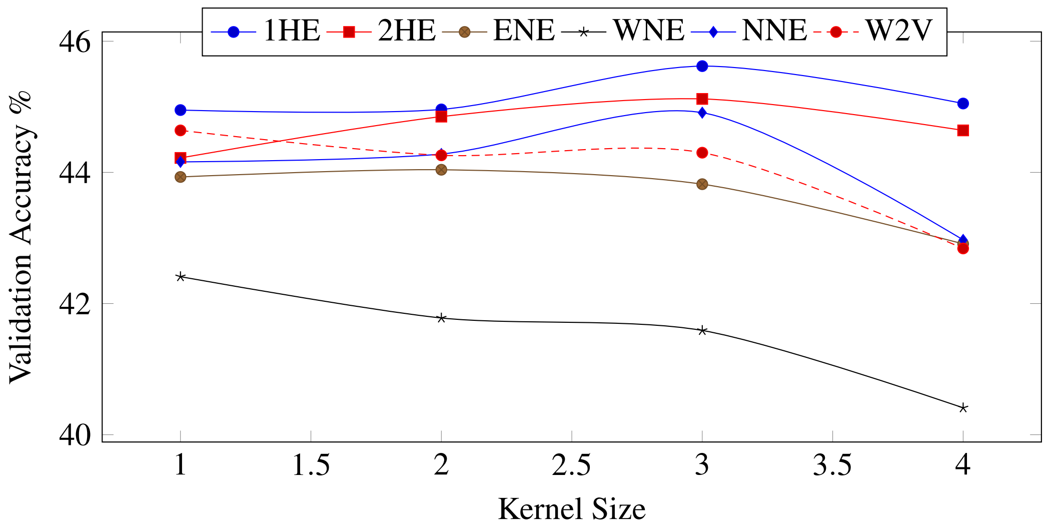

Figure 6.10: CNN with 1 hidden layer and 2000 hidden units. Max values: **1HE**: 3 = 45.62%, **2HE**: 2 = 45.12%, **NNE**: 3 = 44.91%, **W2V**: 2 = 44.64%, **ENE**: 2 = 44.04%, **WNE**: 1 = 42.41%.

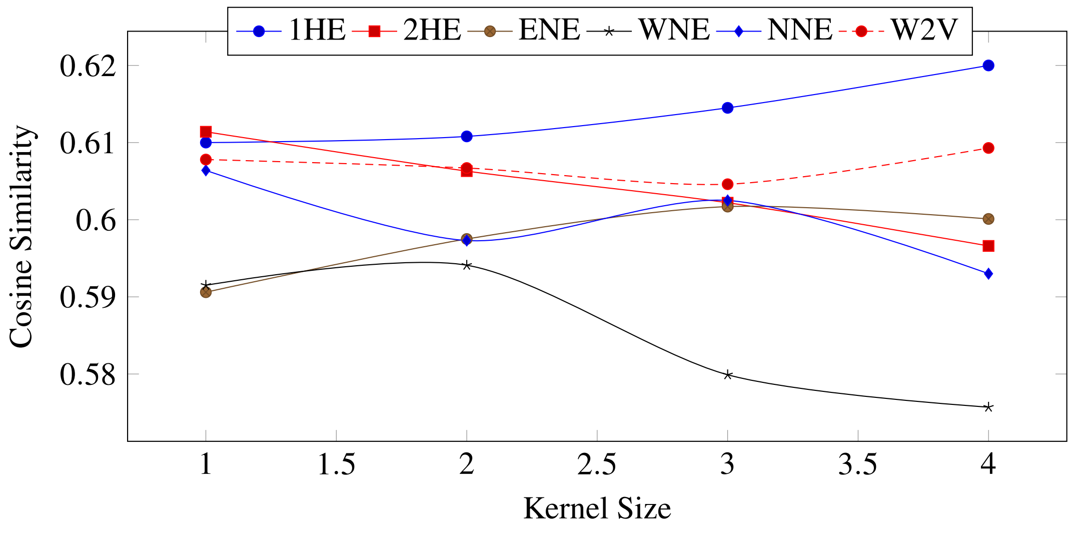

Figure 6.11: CNN with 1 hidden layer and 2000 hidden units. Max values: **1HE**: 4 = 0.62, **2HE**: 1 = 0.6114, **W2V**: 4 = 0.6093, **NNE**: 1 = 0.6064, **ENE**: 3 = 0.6017, **WNE**: 2 = 0.5941.

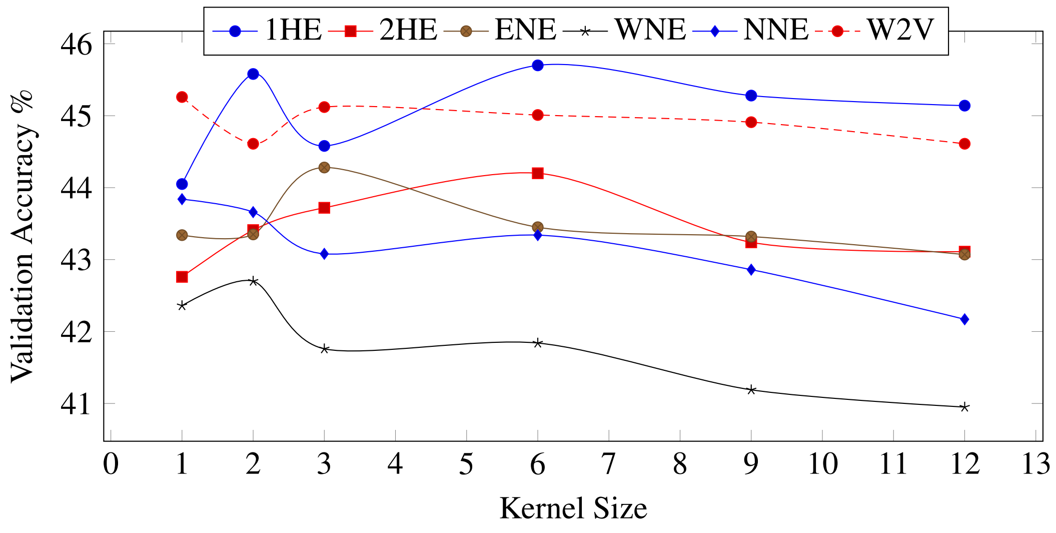

Figure 6.12: CNN with transposed input, 1 hidden layer, and 2000 hidden units. Max values: **1HE**: 6 = 45.7%, **W2V**: 1 = 45.26%, **NNE**: 1 = 43.84%, **ENE**: 3 = 44.28%, **2HE**: 6 = 44.2%, **WNE**: 2 = 42.7%.

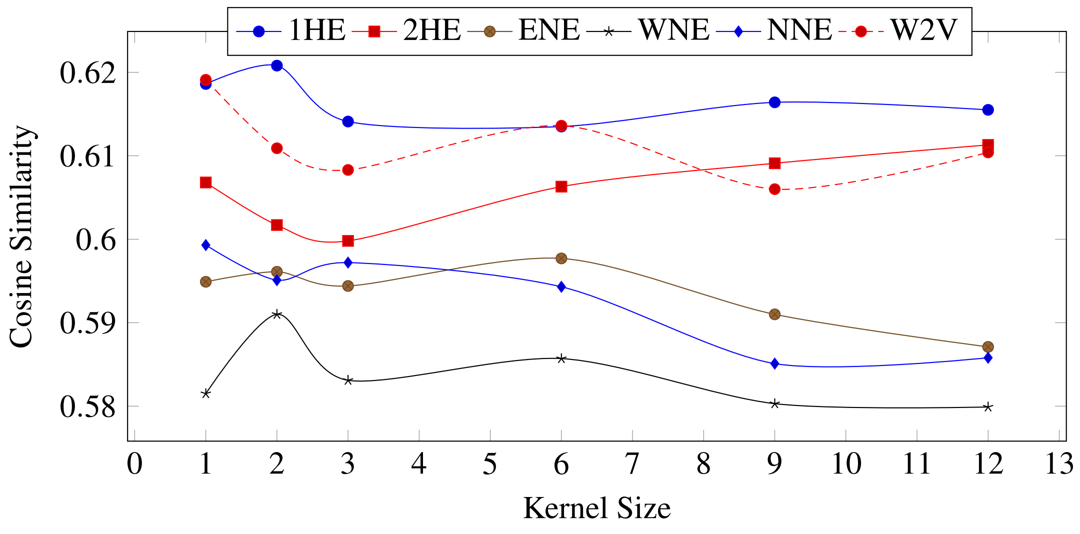

Figure 6.13: CNN with transposed input, 1 hidden layer, and 2000 hidden units. Max values: **1HE**: 2 = 0.6208, **W2V**: 1 = 0.6191, **2HE**: 12 = 0.6113, **NNE**: 1 = 0.5993, **ENE**: 6 = 0.5977, **WNE**: 2 = 0.591.

6.7 Long Short-Term Memory Network (LSTM)

6.7.1 Set up

The final model with which I explore the different representations is the LSTM Neural Network. LSTMs take chords one-by-one, so there is no need for concatenating the input progression. To make this a fair comparison, however, the LSTM was given only 4 input chords to use during testing (not during training), just like the other models.

PyTorch allows for multiple LSTM layers to be used, however, given the marginal benefits gained with more layers in the FNN experiments, I have chosen to use only one layer in favour of quicker experiment run times.

Gers et. al. [10] ran their experiments with forget gate biases between -2 and 2, and so I have done the same, running with values -2, -1, 0, 1, and 2 (Figures 6.15 and 6.16).

6.7.2 Effect of the Forget Gate Bias

The overall trend is I expected; higher forget gate biases yielded higher performance, especially at a value of 1. As the bias pushes the final value of the forget gate closer to 1, the effect of vanishing gradients at the beginning of the training process is minimised. The highest performance was seen again with 1HE at 52.73% accuracy / 0.6462 similarity, making it the best performing model in this work.

Gers et. al. [10] and Jozefowicz et. al. [14] both recommend using a bias of 1 or higher, however, in this case, we see a bias of 0 still performing very well. A forget gate value of 0 run through the sigmoid function means that with every input, the previous output is multiplied by 0.5 (see Figure 6.14), which in a normal situation is a very quickly vanishing gradient. However, because the context window is only 4, this is not really an issue. Compare this to a bias of -1, where after sigmoid we have a value of around 0.27, we can easily see that even with just 4 inputs the vanishing gradient can be a problem, and this is reflected in the results.

I ran additional experiments with bias = 4, confirming that a bias of 1 really is optimal, meaning the results are consistent with Gers at. al. [10] and Jozefowicz et. al. [14] (Figures 6.15 and 6.16).

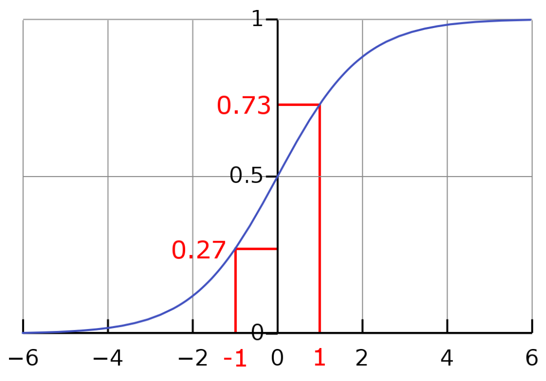

Figure 6.14: Sigmoid function

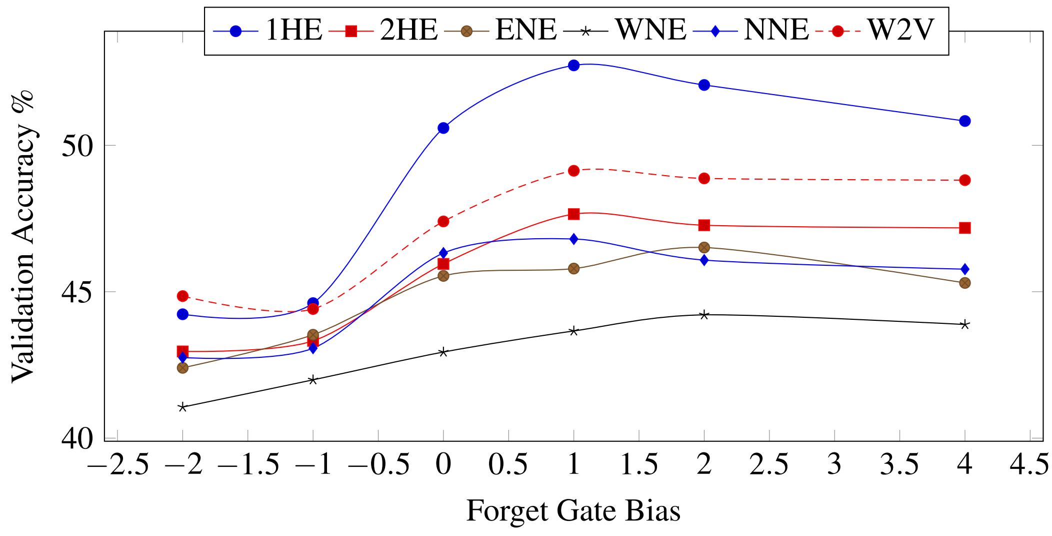

Figure 6.15: LSTM with 500 output units. Max values: **1HE**: 1 = 52.73%, **W2V**: 1 = 49.13%, **2HE**: 1 = 47.65%, **NNE**: 1 = 46.8%, **ENE**: 2 = 46.51%, **WNE**: 2 = 44.21%.

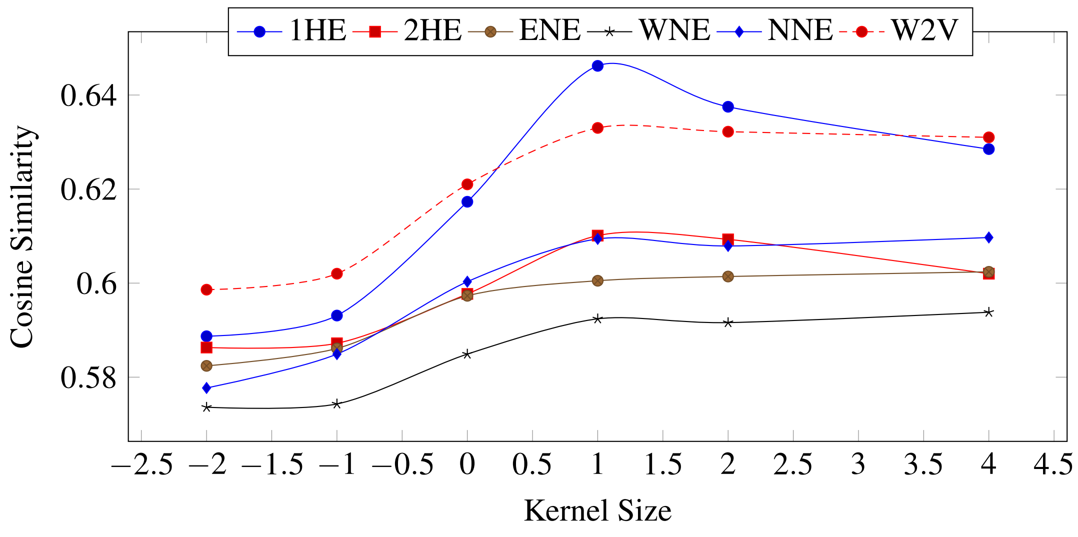

Figure 6.16: LSTM with 500 output units. Max values: **1HE**: 1 = 0.6462, **W2V**: 1 = 0.633, **2HE**: 1 = 0.6101, **NNE**: 1 = 0.6094, **ENE**: 4 = 0.6024, **WNE**: 4 = 0.5938.

Chapter 7

Discussion and Conclusions

7.1 Generalisation

In section 6.5.3 for the FNN results, I noted how my models continued to perform well in the validation accuracy, even as the training accuracy approached 100%. This was unexpected, since neural networks usually decrease in validation accuracy as they start to memorise the training data rather than being able to generalise. **This lead me to the conclusion that for this particular dataset and task, memorising the training data as much as possible is a good, if not optimal strategy.** This makes sense when you consider what the dataset is, and what the task is.

The dataset is composed of songs that are all from one genre, which means it is quite likely that the same musical idea can be found in separate songs. It may even be that some of these ideas are part of the definition of this genre. The task is to predict the next chord in a sequence of chords, and if musical ideas are often shared between songs, it is quite likely that the chord you are tasked with to predict follows a pattern that you have already seen before.

Because this task is more of a ”learn to memorise” task than a more traditional learning task, it favours those networks that have had lots of time to plagiarise and reproduce riff and phrases from songs in the dataset. To look at this from a different angle, anything that the model produces that does not follow the style, is, by definition, not of that style, and cannot possibly be used to predict how a song in that style continues.

7.2 Chord Representations

I was not able to improve on W2V or the state of the art 1HE with either of my proposed representations 2HE or NNE, however, I was still able to improve slightly on ENE with both at around 1-2%. 2HE and NNE were both designed to build on ENE by splitting the chord root and chord components into separately normalised parts, and the results showed that doing this increases the performance of my neural networks for chord generation. A possible explanation for this is that the chord root is arguably the most important piece of information about a chord, so separating it from the other parts makes the chord more easily understood by the network.

Regarding W2V, the performance was lower than I expected. Although W2V was trained specifically to be optimal for the FNN model, it still fell behind 1HE consistently, though it was always close behind. Still, W2V has proven itself to be a strong choice for chord representation, which is a contrast to how little it is used in the current literature.

WNE on the other hand, performed consistently worse than any other representation. It is not used very often in the literature, but it is only a slight variant of ENE which is very popular. WNE essentially condenses the information present in ENE to fit all notes within a single octave rather than the two octaves in ENE. I hypothesised that this could improve performance, as the information is compacted and simplified, but in doing so you make the chord root potentially ambiguous. We have learned from 2HE and NNE that making the chord root explicit is a good way to increase performance, which could explain why WNE performed so poorly.

I recommend using the One-Hot Encoding (1HE) over any other representation, as it has been consistently the best performing representation in almost all situations. Chord Embedding (W2V) is also a strong second option; however, further research with more data is required to determine if it can surpass 1HE.

7.3 Evaluation Metrics

For each of my experiments, I have evaluated the models using both the accuracy-based and the similarity-based metric. In general, I found both metrics to follow similar trends across different models and model parameters. I argued in section 3.3 that a similarity-based metric, such as the word embedding-based method I used, is more representative of human evaluation than a simpler accuracy-based approach. **I recommend that both accuracy- and similarity-based methods be used for tasks where it is possible to use word embeddings, just as I have in mine, as it will provide more insight into the true performance of the models.**

7.4 Future Work

I mentioned in section 5.2.7 that the Chord2Vec paper [20] explores two other chord embedding models, based on the Autoregressive Distribution Eliminator [17] and the Sequence-to-Sequence model [33]. These models would be great explorations in the future to create better chord embeddings, especially the Sequence-to-Sequence model, as the authors found this to perform the best.

Generative Adversarial Networks (GANs) are machine learning frameworks that have been used a lot in the literature with a lot of success. It would be interesting to see how much of an improvement this would add to the models I have explored in this work.

Another piece of future work is to build an interactive interface for the best model in this work, the LSTM, where users can input their own progressions as a start, or generate a new progression from scratch[1].

[1] I will be working on this and publishing the app to https://felixwu.me/chords

Bibliography

[1] Charles Ames. The markov process as a compositional model: A survey and tutorial. *Leonardo*, pages 175–187, 1989.

[2] Thierry Bertin-Mahieux, Daniel P.W. Ellis, Brian Whitman, and Paul Lamere. The million song dataset. In *Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011)*, 2011.

[3] Gino Brunner, Yuyi Wang, Roger Wattenhofer, and Jonas Wiesendanger. Jambot: Music theory aware chord based generation of polyphonic music with lstms. In *2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 519–526. IEEE, 2017.

[4] Ke Chen, Weilin Zhang, Shlomo Dubnov, Gus Xia, and Wei Li. The effect of explicit structure encoding of deep neural networks for symbolic music generation. In *2019 International Workshop on Multilayer Music Representation and Processing (MMRP)*, pages 77–84. IEEE, 2019.

[5] Keunwoo Choi, George Fazekas, and Mark Sandler. Text-based lstm networks for automatic music composition. *arXiv preprint arXiv:1604.05358*, 2016.

[6] Hang Chu, Raquel Urtasun, and Sanja Fidler. Song from pi: A musically plausible network for pop music generation. *arXiv preprint arXiv:1611.03477*, 2016.

[7] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017.

[8] Kemal Ebcioğlu. An expert system for harmonizing four-part chorales. *Computer Music Journal*, 12(3):43–51, 1988.

[9] Douglas Eck and Juergen Schmidhuber. A first look at music composition using lstm recurrent neural networks. *Istituto Dalle Molle Di Studi Sull Intelligenza Artificiale*, 103:48, 2002.

[10] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. 1999.

[11] Christopher Harte, Mark B Sandler, Samer A Abdallah, and Emilia Gómez. Symbolic representation of musical chords: A proposed syntax for text annotations. In *ISMIR*, volume 5, pages 66–71, 2005.

[12] Andrew Horner and David E Goldberg. Genetic algorithms and computer-assisted music composition. In *ICGA*, pages 437–441, 1991.

[13] N P Johnson-Laird. Jazz improvisation: A theory at the computational level. *Representing Musical Structure*, 1991.

[14] Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever. An empirical exploration of recurrent network architectures. In *International conference on machine learning*, pages 2342–2350, 2015.

[15] Slava Katz. Estimation of probabilities from sparse data for the language model component of a speech recognizer. *IEEE transactions on acoustics, speech, and signal processing*, 35(3):400–401, 1987.

[16] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[17] Hugo Larochelle and Iain Murray. The neural autoregressive distribution estimator. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 29–37, 2011.

[18] Omer Levy and Yoav Goldberg. Dependency-based word embeddings. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 302–308, 2014.

[19] Hyungui Lim, Seungyeon Rhyu, and Kyogu Lee. Chord generation from symbolic melody using blstm networks. *arXiv preprint arXiv:1712.01011*, 2017.

[20] Sephora Madjiheurem, Lizhen Qu, and Christian Walder. Chord2vec: Learning musical chord embeddings. In *Proceedings of the constructive machine learning workshop at 30th conference on neural information processing systems (NIPS2016), Barcelona, Spain*, 2016.

[21] Huanru Henry Mao, Taylor Shin, and Garrison Cottrell. Deepj: Style-specific music generation. In *2018 IEEE 12th International Conference on Semantic Computing (ICSC)*, pages 377–382. IEEE, 2018.

[22] Ryan A McIntyre. Bach in a box: The evolution of four part baroque harmony using the genetic algorithm. In *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*, pages 852–857. IEEE, 1994.

[23] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[24] Olof Mogren. C-rnn-gan: Continuous recurrent neural networks with adversarial training. *arXiv preprint arXiv:1611.09904*, 2016.

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

[26] François Pachet and Pierre Roy. Formulating constraint satisfaction problems on part-whole relations: The case of automatic musical harmonization. In *Proceedings of the 13th European Conference on Artificial Intelligence (ECAI 98)*, pages 1–11, 1998.

[27] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[28] Jeff Pressing. Nonlinear maps as generators of musical design. *Computer Music Journal*, 12(2):35–46, 1988.

[29] Colin Raffel. *Learning-based methods for comparing sequences, with applications to audio-to-midi alignment and matching*. PhD thesis, Columbia University, 2016.

[30] Radim Řehůřek and Petr Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010. ELRA. `http://is.muni.cz/publication/884893/en`.

[31] Tobias Schnabel, Igor Labutov, David Mimno, and Thorsten Joachims. Evaluation methods for unsupervised word embeddings. In *Proceedings of the 2015 conference on empirical methods in natural language processing*, pages 298–307, 2015.

[32] Mark J Steedman. A generative grammar for jazz chord sequences. *Music Perception: An Interdisciplinary Journal*, 2(1):52–77, 1984.

[33] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[34] Technimo. iReal Pro. `https://irealpro.com/`, 2020.

[35] Jos Van Der Westhuizen and Joan Lasenby. The unreasonable effectiveness of the forget gate. *arXiv preprint arXiv:1804.04849*, 2018.

[36] Geraint A Wiggins, George Papadopoulos, Somnuk Phon-Amnuaisuk, and Andrew Tuson. *Evolutionary methods for musical composition*. University of Edinburgh, Department of Artificial Intelligence, 1998.

[37] Wikipedia. Jazz harmony — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Jazz_harmony`, 2020.

[38] Wikipedia. List of jazz standards — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/List_of_jazz_standards`, 2020.

[39] Terry Winograd. Linguistics and the computer analysis of tonal harmony. *journal of Music Theory*, 12(1):2–49, 1968.

[40] Li-Chia Yang, Szu-Yu Chou, and Yi-Hsuan Yang. Midinet: A convolutional generative adversarial network for symbolic-domain music generation. *arXiv preprint arXiv:1703.10847*, 2017.

[41] David Zicarelli. M and jam factory. *Computer Music Journal*, 11(4):13–29, 1987.