Chord Generation

Neural Systems

Berkay Isik, Jacky Choi, Mathis Lindner, Moritz Fechner, Theo Rodde

ETH Zürich

1 Introduction

After thinking about many different generation ideas that were all more language oriented, we came up with the idea of creating pieces of music: an international language. Music is related to languages in that words are closely emulated by chords or tones. The importance of the order in the structure makes many concepts of natural language processing applicable to chord generation.

There are many applications to this project: from creating free open source music for people to use as background music, to using the generated chords and melodies to inspire artists for a new song when they have some sort of creativity low. It could be integrated in some music software to create a midi chord progression that follows rules that are set beforehand, such as the tempo, how "randomised" the melody should be and time signature.

2 Data Set

After choosing what we wanted to generate, we had to browse the internet for the most well adapted data sets for our purpose. After a few miss-adventures such as going using this data set, based on the hootheory website, we found ourselves using the POP909 data set that gathers 909 Pop songs and their chord progressions.

2.1 Hooktheory data set

We ran into a few problems with the data set based on the hooktheory dataset, but the main issues were that, too many chord sequences ended almost instantly and that the next chords were originally only based on the last played chords and not just the preceding chord: we would have needed 4 transition matrices or combine them into one by weighting them in a fair way. You can clearly see on the this website how many chords do not have a follow up one. These issues could have been fixed, but since we found a data set that fits our needs better , we went with that one.

2.2 POP909 data set

Our final choice was the POP909 dataset by[6]. This dataset was published with a MIT License and exceeded our criterium of simply having a reasonable amount of songs with MIDIs containing a track having all the chords. It consists of 909 pop songs and each of the pop songs has a MIDI which contains a piano track with the chords of the song. Pop songs being particularly well suited as they are known for their catchy and simple chord sequences. Furthermore the dataset has also the chords of each MIDI extracted using two different algorithms delivering the chord names and their beginning and ending time in the song. The algorithms were programmed to extract many chord qualities e.g. inversions and suspended chords. The two algorithms had matched root notes for more than 75% for more than 800 songs yet there were songs were it was below 40%[6].
All songs of the dataset were composed of 320 different chords. The dataset provided information about the arrangement of chords and the duration of each chord time window. Breaks were labelled with an own N chord. For the analysis, the breaks were summarized to one N chord.
The frequency of the cords varied widely and chords with more than 5000 appearances faced chords with 1 appearance. It is worth mentioning that the chords of high frequency were less complex than the chords with rare appearances. The most appearances were counted for G:maj, whereas C:minmaj7 had only one.
The matter of merging chords in subgroups to simplify the sequence generation was not tackled by us, but could likely be a matter for later projects anyway if the statistic evaluation is hampered.

3 Chord Generation

The next task we faced was choosing suitable sampling methods for the data set we are using. Motivated by the papers [2, 5] we decided to utilize the Markov model to generate chord progressions and compare them with the sequences generated using the hierarchical model described in the paper [1].

3.1 Markov Model

Probability theory is a field of study in mathematics, which revolves around how probabilities interact with one another. When probability theory is applied to mathematically model random systems or phenomena, this method is labeled as a stochastic process. A stochastic process is a time-dependent family $(X_t)_{t \in T}$ of a random variable X where t is a time index belonging to a parameter set T [3]. A stochastic process $(Z_n)_{n \in N}$ with a discrete state space $\mathbb{S}$ is said to have the Markov property if, for all $n \geq 1$, the probability distribution of Z_{n+1} is determined by the state Z_n of the process at time n, and does not depend on the past values of Z_k for $k \leq n-1$ [3]. Therefore, a Markov process is a stochastic process whose future state depends only on the current state. Consecutively a Markov chain is a discrete-time Markov process with a finite set of states[2]. Mathematically for all $n \geq 1$ and all $i_0, i_1, ..., i_n, j \subset \mathbb{S}$ we have:

$$\mathbb{P}(Z_{n+1} = j | Z_n = i_n, Z_{n-1} = i_{n-1}, ..., Z_0 = i_0) = \mathbb{P}(Z_{n+1} = j | Z_n = i_n) \tag{3.1}$$

Algorithm 1: Markov Generation

```
Result: n sequences of length k
Inputs: n, k, TransitionMatrix, MarkovSpace;
sequences ← [ ] ;
for i in range n do
    sequence ← [ ];
    currentstate ← ['start'];
    nextstate ← [' '];
    for j in range k + 1 do
        probabilities = TransitionMatrix[currentstate];
        nextstate = probe(MarkovSpace, probabilities);
        sequence.add(nextstate);
        currentstate = nextstate;
    end
    sequence.remove[0]//remove unnecessary 'start' string from array;
    sequences.add(sequence);
end
return sequences;
```

3.2 Transition Matrices

In our implementation, the probabilities between the states will be treated as time homogeneous. That is, the probability $\mathbb{P}(Z_{n+1} = j | Z_n = i)$ is independent of time $n \in N$[3]. The transition probabilities can be modeled in an $\mathbb{S} \times \mathbb{S}$ matrix. This is called the transition matrix of a Markov chain:

$$[P_{i,j}]_{i,j \in \mathbb{S}} = [P(Z_1 = j | Z_0 = i)]_{i,j \in \mathbb{S}}$$

Using the POP909 data set we have generated two transition matrices, a part of which can be seen in Table 1 and Table 2.

	A:7	A:7/3	A:7/5	...
A:7	0	0.021978022	0	...
A:7/3	0	0	0.076923077	...
A:7/5	0	0	0	...
...	...	...	...	...

Table 1: Excerpt from transition matrix for the first-order Markov and Hierarchical Model

		A:7	A:7/3	...
Ab:maj	Ab:sus2	0.043478261	0	...
Bb:maj	Bb:min	0.018181818	0	...
E:min	A:maj	0	0.006116208	...
...	...	...	...	...

Table 2: Excerpt from transition matrix for the second-order Markov Model

These transition matrices show the probability that the chain will move from one state to the next one. In the Table 1, current states are displayed by the row tags, whereas the future states are displayed by the column tags. In the Table 2, the current state is comprised of two subsequent chords in the sequence. Accordingly we have two row tags for each row.

We decided to create the second transition matrix, which is used with the second-order Markov model to generate chord progressions, because a certain detriment of using first-order Markov chains was mentioned in a paper [4]. First-order Markov chains mainly do not produce ”phrased” passages that can be encountered in pieces composed by humans.

3.3 Sequence Generation

As discussed earlier, we implemented two sequence generation methods to create our chord progressions. The first one is the append-based Markovian sequence generation. In this method we start by randomly picking an initial state from the state space. Using this initial state we sample new states, which we then append to the end of our sequence. The exact algorithm we used can be seen under the Algorithm 1. The second method we used is the substitution-based hierarchical sequence generation. In this approach we also start by randomly picking an initial state from the state space. Afterwards we sample a number of states according to the branching factor using the initial state. Next, we substitute this initial state with the freshly sampled states. This process is executed until we reach a predefined depth, where the algorithm stops. The figure 1 shows an overview of how these two methods generate sequences.

3.4 Chord Extraction for MIDI Generation

To create playable MIDIs the Python library “EasyMIDI” was used. For creating MIDIs from the generated sequences containing the chord names, it was decided to create a Python dictionary containing the chords for “EasyMIDI” to the corresponding chord names as keys. Overall three attempts have been tried. The first attempt was to create a simplified version of the chords. As creating every chord of the dataset would be quite cumbersome. The basic major and minor chords for every note at the fourth octave have been defined using “EasyMIDI”. Chords of the dataset then have been reduced to these chords, yielding well sounding sequences which were not accurate representations of the generated sequences.
The second attempt was to extract the chords from the MIDI files using the extracted chords and their corresponding timestamps from the data set for the song. MIDI files can be processed using the “Mido” library. For each song the piano track containing the chord sequences has been extracted and split into time intervals specified in the chord files. Effectively always the last occurrence of the chord has been saved in

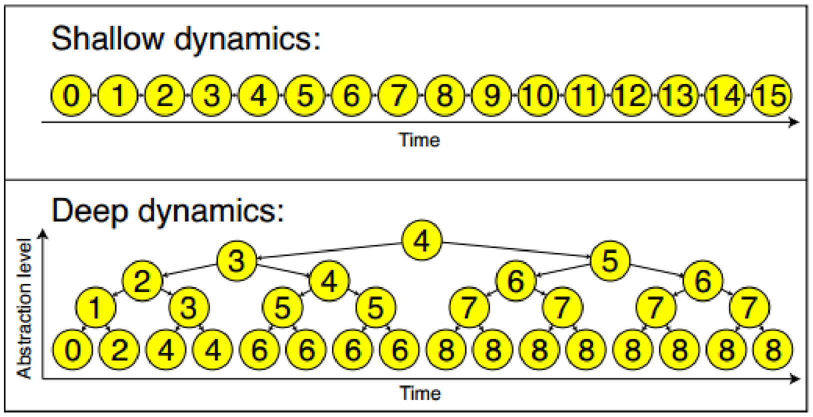


Figure 1: Difference between sequences sampled by Markov model and hierarchical model

the dictionary, so that the chords of the last songs were saved. The notes from these intervals have then been saved to the dictionary to the corresponding chord. As the chords were extracted from different songs, the chords in the dictionary had different octaves and different velocities which did not yield well sounding results. As “EasyMIDI” was more user-friendly than “Mido”, the notes of the chords have been extracted and saved to the dictionary for each chord using “EasyMIDI”, standardizing the octave for each note to the fourth and the velocity to the default. This sounded a little bit better but as chords stretch over octaves it was also not a precise representation. One approach was to set the lowest note to the fourth octave and then setting the higher notes with the same distance as before above it. Again this did not yield satisfying results and also changing the base octave did not improve it. It was notable that in many chords unfitting notes were present, often in a high octave. To filter these notes out of the chords, the strategy to save the notes of the last song where the chord occurs was adjusted. Now the group of notes was used where the distance between the highest and the lowest note was the lowest. However this did not solve this issue either.
One reason for the bad results extracting the chord from the MIDIs using the chords and their timestamps from the data set could be because of notes between the chords becoming part of the chord in the extraction process, distorting the chords. Furthermore according to[6] some of the audio is out of tune and some audio has complicated sound effects.
To solve the problem, we used some web-scraping to find the notes that made up the chords from the scales-chords website, by entering the chord string in the search bar and looking for the table that included the note names. The websites search bar also automatically auto corrected the chord names.

3.5 Melody

To generate a melody that would fit the chord sequence, we took the notes that make up the chords, stored them in an array, elevated them to a higher octave and randomly used one of the chords to be played every fourth of a beat. In this algorithm we included the option to have any time signature. To make the sequence more interesting we also added a "silent note" to leave some gaps in the melody.

```
Algorithm 2: Chord with Melody
Result: MIDI file
Inputs: chordSequence, timeSignature, tempo, chordDictionary
 //chord dictionary holds EasyMIDI objects, time signature for the notes(1, 1/2 1/4...);

song ←EasyMIDI.song() //empty song object;
song.setTempo(tempo);
chordTrack ← [ ] ;
noteTrack ← [ ] ;
for chord in chordSequence do
    chordObject = chordDictionary[chord].setDuration(1) //set length to 1 bar;
    chordTrack.add(chordObject);
    notes ← Chord.getNotes() //get notes that make up the chord;
    notes ← elevateNotesAnOctave(notes).setDuration(timeSignature);
    for i in range 1/timeSignature do
        noteTrack.add(getRandom(notes)) //get a random note from the elevated notes, including a "silent not

    end
end
song.add(chordTrack);
song.add(chordTrack);
EasyMIDI.write(song);
```

We could create more complicated patterns using music theory that explains chord and note that harmonise based on this function. We could also add more melody and chord tracks to the song, but we kept this algorithm to have a firm base.

4 Generated Pieces

The links for the music files can be found on this git repository.

There are four different files: The file called "Hierarchical Model.ogg" used the Hierarchical Model, you can hear why we didn't go further into this kind of generation, the chords just do not sound good one behind the other. The one called "only chords.mp3" only includes 4 chords that were directly translated from the generated Markov model to a midi to an mp3. The one called "chord and melody.mp3" used an algorithm close to Algorithm 2, but without the randomisation of the sequence of the notes. The exact progression can be visualised on Figure 2 "long sequence.mp3" used Algorithm 2 and is, as the name says, longer than the other sequences. It shows how well the the chords can follow one another while sounding good. The layering with notes also sounds interesting enough to keep the listener interested.

5 Discriminator

Given two pieces of data, one real and fake, a discriminator is tasked with finding out which of the two is the real data. In our case we want it to distinguish between the output of the Markov model and the real data. Moreover, when the discriminator is unable to differentiate between the two, our Markov model did a good job of generating a realistic sequence.

Before we discuss the implementation of the discriminator, we have to explain the motivation behind the

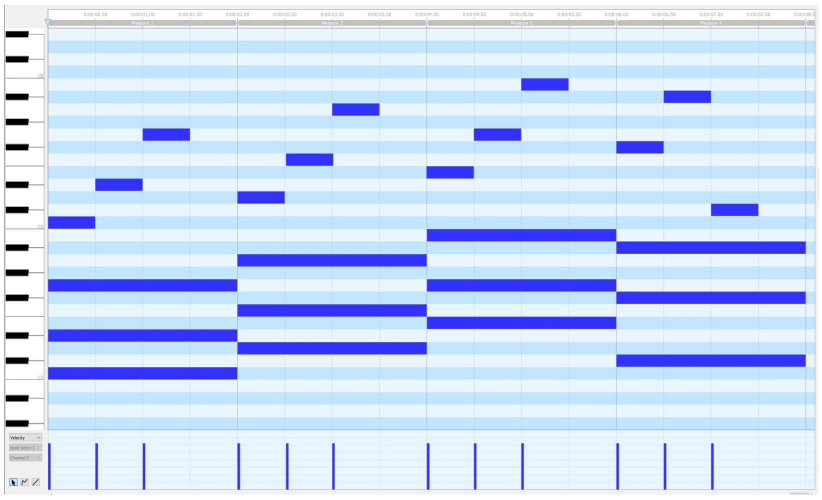


Figure 2: Generated chord and melody

discriminator. Since we talk about music here, a subject that is highly subjective and polarizing, if we want to improve our chord progressions, we have to find an objective evaluation score. The importance of such an evaluation is stressed by the fact that we did not have any knowledge of music theory with which we could have help us along the way. So going forward, if we wanted to further pursue unsupervised learning, a discriminator will be of tremendous help.

5.1 Model

Inspired by several papers that implemented a generative adversarial network for text generation, we opted for a similar architecture for the discriminator, since we saw several similarities between chords and words. However, we chose to implement our problem as a classification task. We labeled the real data with 1, leaving us with the question of how we create our fake data. The network work was implemented in Python, using the Keras API.

As for the fake data, we tried two approaches. Our first naive approach was to take all the distinct chords of the real dataset and just randomly sample from that pool of chords with uniform distribution, until we yield a sequence of desirable length. Our second approach was to sample weighted unigrams and bigrams from our pool of distinct chords. Another factor to take into account was the length of our chord sequences. At first our goal was to generate sequences of length 4. However, since we noticed that the convolution model would not be very applicable for that we settled on lengths 15 and 50.

Inspired the SeqGAN[7], we choose a similar architecture. Our network includes an embedding layer which serves as a dimension reduction of our feature vectors. This will be followed by layers of 1D-convolutions with subsequent batch normalization. Afterwards, we apply a max-pooling layer and flatten the obtained vector and we put it through a fully-connected layer to finally get out output prediction.

5.2 Preprocessing and Training

Before we could even train the network, we had to preprocess our data into a suitable format. Starting out with the .csv-file, which we read into Python as a dataframe, we first reshaped the dataframe to have rows of our desired length. Furthermore, we took the ”songs” (rows) and chopped them into chord sequences of fixed length and each of these sequences will become a row in the new dataframe. Afterwards, we have to transform each row of into a string separated by empty spaces. This format is necessary for us to be able to

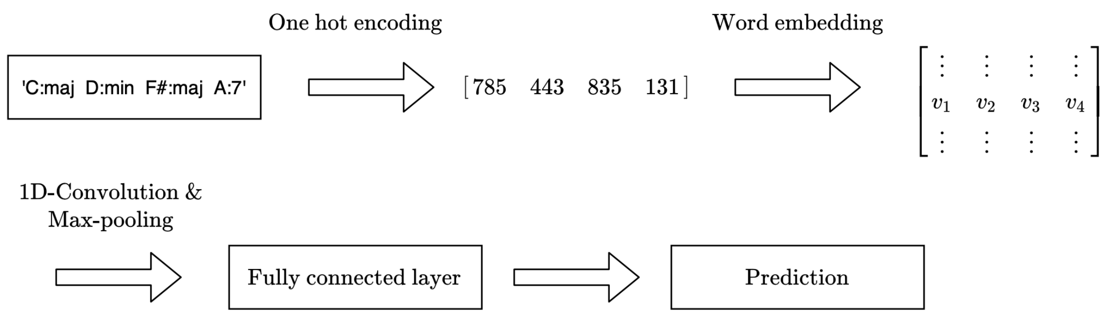


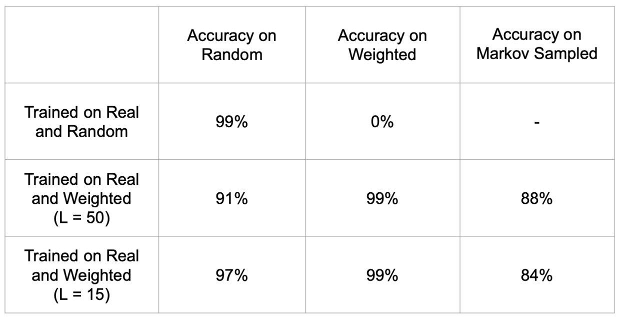
Figure 3: Workflow of discriminator

use the one hot encoding provided Keras, which will be the final format of our input to the network.

To train the network, we implemented a 80/20 train-test split. As for the fitting parameters, we used the Adam optimizer and mean squared error. The model converged after only three epochs of training.

5.3 Findings

Our first naive attempt to learn from real and random data was abandoned quickly, after we received an accuracy of 0% when we wanted to predict the output of weighted samples. This might be an indicator that our network has the ability to pick up important transitions in real data, which it then recognized in the weighted samples.



	Accuracy on Random	Accuracy on Weighted	Accuracy on Markov Sampled
Trained on Real and Random	99%	0%	-
Trained on Real and Weighted (L = 50)	91%	99%	88%
Trained on Real and Weighted (L = 15)	97%	99%	84%

Figure 4: Validation accuracy

After changing the training data to a combination of weighted bigrams and unigrams, the accuracy on the Markov sequences were 88% and 84% for length 50 and 15 respectively. To conclude, our Markov model is not yet ideal, since on paper, its structure is still easily distinguishable from real data. However, the generated MIDI-files show that it is not that unpleasant to the ear as one would think.

6 Conclusion

The results are really promising in reference to the possible applications of the project. We did manage to create chord progressions that sound good with melodies on top that fit well.

The discriminator showed that from a structural point, the generated chords are still quite distinguishable from the real sequences. Furthermore, in the future we could use the feedback from the discriminator to maybe alter and improve the Markov transition matrix. This can be combined with reinforcement learning.

We could work on creating songs following a classic pop song pattern with an introduction, a chorus, verses, bridges and an outro that uses the transition matrix and algorithm 2. Tied to the classic pop song pattern, we could further explore different sequence generation techniques like the hierarchical model or the hybrid model.

References

[1] H. W. Lin and M. Tegmark. Critical behavior in physics and probabilistic formal languages. *Entropy*, 19(7), 2017.

[2] E. J. Linskens. Music improvisation using markov chains, 2014.

[3] N. Privault. *Understanding Markov chains: examples and applications.* Springer, 2018.

[4] C. Roads. *Computer Music Tutorial.* 1996.

[5] A. Van Der Merwe and W. Schulze. Music generation with markov models. *IEEE MultiMedia*, 18(3):78–85, 2011.

[6] Z. Wang*, K. Chen*, J. Jiang, Y. Zhang, M. Xu, S. Dai, G. Bin, and G. Xia. Pop909: A pop-song dataset for music arrangement generation. In *Proceedings of 21st International Conference on Music Information Retrieval, ISMIR*, 2020.

[7] L. Yu, W. Zhang, J. Wang, and Y. Yu. Seqgan: Sequence generative adversarial nets with policy gradient, 2017.