Generating Chord Sequences with Deep Learning

Evan Kranzler

April 5, 2018

Introduction

Much has been written about using Deep Learning to learn and predict text. It's been utilized to recreate Shakespeare, simulate subreddits, recreate Beethoven, and even write strange, incomprehensible screenplays. The recurring theme in each of these situations is that one starts with text, each character is encoded individually, and the neural network learns them as sequences. But what if instead of feeding the network characters, we fed it something with more breadth? This is the idea behind this project.

How do chords work?

Typical western music uses notes from what is called the *chromatic scale*. This is a sequence of notes, typically denoted using the letters A through G, which are separated by units called *half-steps*. A chord is when three or more notes are played simultaneously.

For example, we'll look at the A major chord. The A major chord consists of the notes A, C# and E. This is because if we wrote out an A major scale, it would consist of the notes A, B, C#, D, E, F#, G#, A, and the first, third, and fifth notes of the scale are what make up the A major chord.

Observe that we could play a given note in more than one octave, which is known as a *voicing*. However, we must keep the lowest note the same. So if we take our A major chord and move the C# an octave higher, the actual underlying properties of the chord will remain unchanged. If we move the A an octave higher instead, the lowest note will be the C# and we will have what is known as an *inversion*.

What's important to take away from this is that a chord has two important characteristics: the notes it contains, and which of those notes is the lowest, or *root* note. This means we can describe chords as a vector. Let's take another look at A major:

First, we define our lowest note to be C. This is an arbitrary choice. We will have a 24-dimensional vector, with the first 12 dimensions defining the root and the next twelve defining the notes in the chord. Our A major chord will be represented as

$$[0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,1,0,0,0,0,0,1,0,0]$$

The tenth place represents the note A, and the thirteenth, sixteenth and twenty-second places represent C#, E and A respectively. This format allows our notes to be put out of order on the second half, as we only care what the lowest note is.

Processing lead sheets

Using "The Imaginary Book", we have a database of jazz standards with their associated chord progressions. The chord portion of each file is formatted as follows (we'll use Blue Bossa as our example):

```
Cm69 | / | Fm9 | / |
Dm7b5 | G7#5#9 | Cm69 | / |
Ebm9 | Ab13 | DbM9 | / |
Dm7b5 | G7#5#9 | Cm69 | Dm7b5 G7#5#9 |
```

The way we interpret this is that the | symbol denotes the separation of measures. The first measure has a Cm69 chord, then the / in the next measure means to repeat the previous measure. Additionally, the last measure on the last line includes two chords, meaning that each chord gets half the measure. This means that our time series needs a smallest unit, which is given by the shortest amount of beats given to a chord in a given tune.

When parsing these, we split first by the | delimiter and have a list of measures. We split each measure by spaces, and end up with a list of lists. Here's where it gets tricky- we want to distribute chords in such a way that each one gets the appropriate amount of beats. In order to do this, we look at the largest amount of chords in a measure, then use that with the time signature to properly space things out. There are some annoying situations we have to deal with, but when we're done, Blue Bossa looks like this:

```
Cm69 Cm69 Cm69 Cm69 Cm69 Cm69 Cm69 Cm69
Fm9 Fm9 Fm9 Fm9 Fm9 Fm9 Fm9 Fm9
Dm7b5 Dm7b5 Dm7b5 Dm7b5 G7#5#9 G7#5#9 G7#5#9 G7#5#9
Cm69 Cm69 Cm69 Cm69 Cm69 Cm69 Cm69 Cm69
Ebm9 Ebm9 Ebm9 Ebm9 Ab13 Ab13 Ab13 Ab13
DbM9 DbM9 DbM9 DbM9 DbM9 DbM9 DbM9 DbM9
Dm7b5 Dm7b5 Dm7b5 Dm7b5 G7#5#9 G7#5#9 G7#5#9 G7#5#9
Cm69 Cm69 Cm69 Cm69 Dm7b5 Dm7b5 G7#5#9 G7#5#9
```

The $Cm69$ chord is repeated eight times, four for each measure as the tune is in 4/4. At this point we can convert our chord symbols to our previously described vector notation, and we're done.

Preparing the data

After we've processed all of our lead sheets, we want to transpose each one into each key. This is analogous to rotating and reflecting images before running them through an image classifier, and has the added benefit of expanding the training data twelvefold.

We now have a list of vector sequences. We shuffle this list, then concatenate them into one long sequence, adding eight empty chords in between each one. A single input is a 32-chord sequence, and its target is that same sequence but with the first chord dropped and the next chord added to the end.

Building the network

The network is built using Keras, with the following topology:

- A masking layer which ignores empty chords, to help the network learn where there are breaks.
- Two dense layers, each with 48 nodes. This is hopefully where the network will learn harmonies and create new features. These were previously convolutional layers, but with only 24 input features they were unnecessary.
- Three LSTM layers, each with 1024 parameters, and set to return sequences. This is where the network actually learns how things are ordered, and will be how it learns to predict.
- One single dense layer with 24 nodes for output

Each layer has 50% dropout to help prevent overfitting, and each non-LSTM layer has been made time-distributed, which allows it to better apply to sequential data. The internal dense layers have ReLU as their activation functions, which is fairly standard, and the final layer has sigmoid, with the loss function being binary cross-entropy.

The choice of the activation and loss functions here are what set this model apart from others which use softmax and categorical cross-entropy for character-based one-hot encoding.

Results

This is unfortunately where things become problematic. Ideally, the network should be "stateful", where it maintains a longer-term memory. The issue here is that, when implemented, the network took a prohibitively long amount of time to train. Therefore, statefulness had to be dropped in the interest of time.

Additionally, the traditional way of implementing character-based models involves using the output to sample from a distribution. This is not as clear when they output is intended to have multiple classes. This means that after training the network, trying to generate a sequence leads to something which

either appears completely random, or repeats the same output over and over again.

Conclusions

This project was a mixed bag. One the one hand, a lot was learned about how LSTM networks work and can be used, and a lot of intuition was gained on various aspects of neural networks. At the same time, satisfying results were not achieved and there was not enough time to pursue certain goals. All in all, this project was successful in exploring deep learning.