(Monophonic) Audio to MIDI conversion

This is a brief introduction on how audio-to-MIDI conversion works. Have fun!!

Introduction

Symbolic annotations for audio are useful for many purposes. Humans themselves do that all the time. That is, we do not see people going around saying: "so you play 'bam' then 'dararirada'" (ok, sometimes you do, but in very particular situations). Instead, we tend to say: "play a loud G, then C, D, E, D, C"

One of the first computer-friendly musical symbolic standards was the MIDI (Musical Instrument Digital Interface) protocol, dating back to the early '80s (check https://www.midi.org/). MIDI initially defined a full protocol stack and its corresponding circuitry. However, nowadays, the physical and link layers are progressively changing to more modern standards, such as USB, file streams, and wireless connections. The interesting about MIDI is that is maps a pre-defined set of messages into musically-meaningful elements, like "play note N with velocity V" or "stop playing note N," making it easier to establish communications between devices from different manufacturers.

MIDI messages can be recorded into MIDI files, which are read by devices such as synthesizers or effect controllers. We can then have a MIDI file with the transcription of,

say, a Mozart symphony, and then synthesize it with our favorite timbres and effects.

A solution to make MIDI files is to have notation software (MuseScore, for example) directly export them from a standard score. This type of software usually creates MIDI files with all notes annotated precisely on the beat, and whose velocities are constant, ultimately leading to inexpressive interpretations of the pieces. The other problem with this is that the user (a musician) has to manually transcribe music by ear, which can be very time-consuming.

Another method to make MIDI files is to record performances from digital instruments. This solution mitigates the expressiveness problem of using notation software to the extent of the musician's skills with that particular instrument. However, it could be harder to replicate the expressiveness of instruments that demand a higher physicality: using a keyboard to emulate a piano performance could be fine, whereas the expressiveness of wind, brass, or string instruments could be affected when switching to a keyboard interface.

Finally, another solution could be to have a digital computer read audio files (or audio streams) and directly generate a corresponding MIDI representation. This idea is not novel at all - this topic had its first publishings in work by Piszczalski and Galler (1977, "Automatic Transcription of Music," Computer Music Journal), and by James A. Moorer (1977, "On the Transcription of Musical Sound by Computer" Computer Music Journal).

In the savage late '70s, computers were less powerful, and even these small applications required state-of-the-art hardware to execute. There were no libraries, no standards (remember: MIDI first came out in 1983, and the WAV file standards only appeared in 1991. Also, there was much research on this topic in the last 40 years. As a result, automatically finding symbolic representations to audio can be done much faster and more reliably than decades ago.

Here, we will show how to use Python's librosa to build an automatic solfege-to-midi converter.

How does audio-to-midi conversion work?

Most audio-to-midi converters use the idea that pitched sounds are perceived when we (humans) are exposed to periodic sound waves. The periodicity property makes it

highly useful to represent the waves as Fourier Series, that is, a weighted sum of sinusoidal signals whose frequencies are multiples of a fundamental F0) . The pitch, which allows us to order sounds from bass to treble, is so directly correlated to F0 that many articles on automatic music transcription use the words "pitch" and "F0" indistinctively.

It is possible to detect pitch in small frames of an audio file. This detection yields information that is later combined to find discrete events. Last, we write these events into a MIDI file, which we can render layer with a synthesizer.

Fundamental frequency detection

There are many different methods to detect a signal's F0. They all somehow exploit the idea that pitched signals are periodic. Hence, we can find the fundamental frequencies by summing harmonics, estimating the peaks in the signal's autocorrelation, or finding delays that lead to signal invariance. These methods (and others) have inspired a great amount of scientific word in the past.

More recently, pitch detection was improved using the idea that musical pitches tend to remain constant along time. This idea inspired a probabilistic model that allowed the pYin method to outperform its predecessors in several benchmarks.

pYin is part of librosa since version 0.8.0. It receives audio samples (in an array) as input and analyzes it in known length frames. Librosa's implementation returns the pitch, a voiced flag, and a voiced probability, for each audio frame. The pitch is the estimated F0 in Hz. The voiced flag is True if the frame is voiced (that is, it contains a pitched signal). The voiced probability informs the probability that the frame is voiced.

There is a small catch in the frame length parameter. F0 estimation starts by selecting a subset of the frame and delaying it by some samples. It then proceeds to calculate the squared sum of the difference between the original frame and its delayed version. This squared difference (or residual) has local minima in delays that are multiples of the fundamental period (1/F0).

This procedure can only work if the total frame length is at least twice the length of the largest fundamental period (that is, > 1/F0) that you expect to find. Hence, while smaller frames provide a finer time-domain resolution to F0 estimation, they can lead to missing some bass notes. Luckily, frames of around 23ms (1024 samples at 44100Hz

sampling rate) are enough for most musical notes, and 46ms are enough even for the lowest notes of a piano.

Onset detections

Musical notes usually start at some point. The starting point of a musical note is called onset. While the pitch allows us to differentiate which note is playing, the onset informs when the note starts.

There are many different methods to detect note onsets. Each one of them works best for a different type of signal (for more information, check this tutorial article by Juan P. Bello et al., 2008). Essentially, onset detection works by finding when the contents of a signal frame indicate that something new (in the case of monophonic melodies, "something new" can only mean a new note!) happened since the previous frame.

Librosa has a pre-built onset detector. It fails in more complicated sounds, but it works just fine for monophonic audio. Librosa's onset detector receives as parameters both an audio signal and information on the frame size used to compute the onsets. Using the same framing configurations as in the pitch detection stage, we avoid dealing with different sample rates in the pitch and onset signals.

After just a few lines of code, we have signals that indicate both the pitch and onsets of our audio signal. We could spend some time devising a smart rule set to convert that to discrete symbols, but, instead, we are going to use Hidden Markov Models.

Hidden Markov Models

Theory behind HMMs

There are many online resources to teach Hidden Markov Models (HMMs), including this fantastic tutorial article by Rabiner. HMMs are fundamental to many speech-to-text applications, and their usage in music follows the same ideas.

Markov Chains

We can interpret HMMs as an extension of Markov Chains (MCs), which are probabilistic automata. MCs are used to describe systems that can assume one single state in each discrete time. When time passes, the system assumes a new state according to a probability distribution that depends on its current state.

Henceforth, we can visualize Markov Chains as graphs whose vertices represent states and directed edges represent the probability of transitioning from one state to another. At each discrete time, we randomly select a transition to a new state.

This type of model has been part of music-making since Xenakys in 1955. In the '80s, David Cope proposed to model musical notes as MC states, and estimate transition probabilities by counting occurrences in a symbolic corpus. Cope's idea was to emulate musical style by encoding it into the MC's transition probabilities; that is, a set of transitions can emulate Bach, another set can emulate Mozart, and so on.

HMMs extend the idea of MCs with an observable behavior related to each state. This behavior is called emission. Observable means that the emission is directly measurable somehow.

An HMM example using archery

The components of an HMM can be better understood using an example. Suppose we have two archers. One of them (archer A) is very precise and always hits points within a 10cm radius from the target. The other (archer B) is less precise and hits points within a 20cm radius from the target. In this example, we do not know which archer shot the arrow, but we can observe the distance between the arrow and the target at each time.

Hence, we can model this situation using an HMM in which the emissions are the distances between the arrows and the targets, the states correspond to the archers, and the discrete time passes as they shoot arrows. The power of HMMs in digital signal processing comes from the problem of detecting what archer shot each arrow, given that we have observed the arrows.

Without information on how the archers choose to shoot, the maximum likelihood solution is to assume archer A shot all arrows closer than 10cm to the target, and archer B shot all the others. However, there is a chance that archer B hit closer to the target than archer A.

If we know (for some reason) that if archer B always shoots at least two rounds in a row - and I am exaggerating the probabilities to avoid complicated calculations - we could correct our previous time-independent assumption. This correction would happen, for example, in a sequence like 5, 15, 25, 15. The prior assumption is that the corresponding archers are A, A, B, A.

However, if we know that there is zero chance that archer A shot the third arrow and that archer B always shoots at least two arrows in a row, then we can correct our assumption about the fourth arrow. Our shooter sequence then becomes A, A, B, B.

If the number of archers and the complexity of the assumptions grow, then it is clear that estimating the archer order becomes much more challenging. The good news is, there is a fast algorithm that solves this problem. It is called Viterbi's algorithm, and Librosa 0.8.0 brings an implementation.

In Librosa, Viterbi's algorithm receives as input a transition matrix, in which element A[i,j] is the probability of going from state i to state j at each discrete time. It also receives a prior probability matrix P[i, t], which represents the prior probability related to state i at discrete time t. It then returns both the most likely sequence of states and the posterior probability related to the sequence.

In automatic music transcription, we can observe pitches and onsets and their framings determine the discrete time. Hence, we model musical notes - which are not directly observable - as HMM states. Next, we will build the models.

Musicological Models

A musicological model is a Markov Chain built to emulate musicological characteristics. They are close to David Cope's idea, that is, states correspond to notes, and the transition probabilities encode information related to how we expect notes to transitions between themselves. A musicological model for strictly tonal music, for example, could have a small probability in the transition related to the diminished fifth (V-, or, going from C to Gb), and a greater probability related to intervals of IV or V.

In our musicological model, we do not want to assume anything about our input (at least so far), so we will assume that all state transition probabilities are equal.

The prior probability related to each note depends on the pYin estimate for each frame. If pYin estimated the pitch corresponding to that note, then the prior probability should

be high. Else, the probability should be lower.

We can also account for the detuning of musicians doing unaccompanied solfege. For such, we can assume that pYin estimating a particular pitch can be a sign of the musician's attempt to reach a semitone lower or higher.

Last, we add a silence state, which represents pauses and other unvoiced frames.

Acoustic Models

The musicological model is ambiguous because a sequence of repeated states can be related both to note repetitions and to note sustains. We can solve this ambiguity by modeling each note with an acoustic model.

The acoustic model consists of at least two states. The first one corresponds to the onset of a note, and the subsequent ones model its behavior during the note sustain. The idea is that transitioning to an onset state represents starting a new note, and paths through the sustain states represent that the note is still playing.

In our implementation, we use a two-state acoustic model. The probability of transitioning from the onset to the sustain state is always 1. The probability of remaining in the sustain state is related to our expectancy about the note duration. Each sustain state can also transition to the other onset states according to our musicological model.

Similarly to our use of pYin in our musicological model, the onset detector provides useful information for the acoustic model, as follows. The prior probability of being in an onset state is high if the corresponding frame is an onset (as estimated by the detector); else, the probability is low. The sustain state follows the inverse of this rule.

Estimating priors

So far, we have used general "high" or "low" labels to describe the relationship between observations and prior probabilities. These relationships are related to our confidence in the estimators.

If the estimators (pYin and onset_detect) were correct all the time, then all high/low probabilities would be 1 and 0. These algorithms are reliable, yes, but musicians are

often not so much. Real music can drift in pitch, have some unintentional vibrato, or have artifacts that the onset detector can interpret as a new note.

We could, theoretically, use the Baum-Welch algorithm to estimate optimal parameters for our HMM. However, we do not have a music corpora that models each specific musician's deviation from what we have modeled as an "ideal" performance. For this personal use, it seems more useful to have an adjustable tolerance parameter that can be changed by the musicians using the transcriber.

Writing a MIDI file

Up to this point, we can open a sound file, estimate its pitch and onsets along time, and then execute Viterbi's algorithm to find out the most likely sequence of states. After that, we can loop through the states and find note onsets and offsets, which allows us to create a list containing the onset, offset, and pitch of each note.

Unfortunately, Librosa does not currently implement MIDI file I/O, so we have to use a different library. In this implementation, we used midiutil.

Midiutil writes each note's duration as a multiple of a quarter note. The relative time domain allows changing the tempo when rendering the MIDI file. However, our application is all bound to time in seconds; hence, we need to estimate our original audio file's tempo.

Librosa implements a bpm detector. It receives the audio samples as parameters and returns a bpm in beats-per-minute. The quarter note duration, in seconds, can be directly estimated from the bpm.

After that, we can convert all durations in seconds to multiples of a quarter note and then add them to a MIDI file.

Happy music-making!