Polyphonic Audio Transcription

By Christopher Fuentes

CSCI S-89

Problem Statement

Can we develop a machine learning model that can transcribe recorded audio into a symbolic representation?

Overview

Music can be represented audibly (e.g. via sound recordings) or symbolically (e.g. via sheet music). Like the written word, symbolic audio has no intrinsic sound; rather, it is a set of instructions on what sounds to produce and when. Just as musicians have been performing songs from sheet music for centuries, so too can computers easily translate symbolic music formats into audio formats by applying voicing libraries and simply executing the musical instructions in sequence.

However, the reverse is not true: given an audio recording, neither human nor machine is particularly adept at producing a symbolic representation thereof. Depending on the complexity of the piece, even a well-trained musician can take hours to transcribe a recording into sheet music by hand. My task was to train a machine learning model that could produce a symbolic representation of audio (i.e. a transcription) given only an audio recording as input.

My approach would be to model the problem as a translation task: the audio input is a sequence of signals, and the symbolic output is a sequence of symbols. While I was not able to achieve good results in the end, my experiments and research suggest that better results may be possible to achieve with some modifications to the data processing algorithms and network architecture.

Definitions

Wav format: A file format for recorded audio ("waveform"). The audio is transcoded as a series of 32-bit floating point numbers which describe the physical waveform of the audio. Typically 44,100 floats are used to represent 1 second of recorded audio (note: audio can be recorded at much higher rates such as 196k / second, but will also be recognizable at much lower rates such as 16k)
Midi format: A symbolic file format for audio (Musical Instrument Digital Interface). Midi files contain a set of note instructions in sequence which, when executed by a sound voicing library, would reproduce an audio track. You can think of midi files as a machine-friendly version of sheet music, and in fact it is simple to produce human-readable sheet music from a midi source since they are both 1:1 symbolic representations of the same audio.
Midi event: A single sound (or silence) event in a midi file. This typically has a start time, velocity (similar to volume), program (a way of indicating what instrument should produce the sound), pitch (which note to play), and end time.

Dataset

I used the Musicnet dataset for training. This is a collection of about 300 recordings of classical music as well as corresponding midi transcriptions and text “labels” for every note in the recordings. The dataset ended up requiring a bit of modification, see “Data Preprocessing” below.

Environment Setup

I am using an Apple M1 Pro-based mac running OSX 12.2.1 with 16 gigabytes of ram. The CPU architecture, being Arm64, meant that a lot of library packages were either entirely incompatible or required some awkward workarounds.

Dependencies

Python

I used python 3.8.13 (installed via conda), though I believe 3.9 will also work.

You’ll need to install the following python packages:

```
note-seq==0.0.5
soundfile
tensorflow_text==2.9.0 # ← installed via an arm64 wheel
seqio
```

In addition, I installed the following via conda:

```
tensorflow==2.8.1
librosa
```

The project has a dependency on Magenta’s MT3 library for some of the midi processing. However, it was not possible for me to install the library via pip or even reference it as a git submodule, because the library has an upstream dependency on the Google T5X library, which can not be installed on an Apple M1 chip. Fortunately, the library functions I needed were fairly isolated, so I’ve just imported those files directly and removed any references to unneeded dependencies.

Native Libraries

You'll also need the following native packages for the data preprocessing step (these are available via Homebrew for mac):

```
libsndfile
fluidsynth
```

Finally, you'll need a soundfont (I used "Essential Keys-sforzando-v9.6").

Data Preprocessing

Though the musicnet dataset comes with csv labels for every audio recording, the authors point out that there is a 4% error rate in the timing of the labels which I wanted to avoid somehow. Also, despite the authors' claim that the audio recordings in the dataset are of high quality, as an audio engineer I am not in full agreement with the statement. Lastly, the provided midi transcriptions do not match the length of the provided audio recordings, so they can not be used as-is without some preprocessing.

To solve all of these problems at once, I utilized fluid synth to regenerate the wav files from the midi (the exact opposite of what the network should eventually accomplish). This would provide clean audio recordings that would perfectly match the midi files, which could then be used as the labels for training.

First I reorganized the data into a parallel folder structure: one containing audio tracks, the other containing midi tracks, identically named except for their file extensions.

I've provided a python function to do this, which you can easily invoke:

```
from data_processors import musicnet

musicnet.rename_data(
	audio_dir='/path/to/unzipped/musicnet/audio',
	midi_dir='/path/to/unzipped/musicnet/midis'
)
```

Now we'll regenerate the audio using fluidsynth and the musicnet midi files via a simple bash script:

```
mv data/audio data/original_audio # move the originals out of the way
mkdir data/audio # ensure we have a data/audio directory

soundfont=/path/to/your/sound/font
```

```
for f in $(ls data/midi); do
 wav_filename=$(echo $f | sed s/.mid/.wav/g)
 echo "Converting ${f} to ${wav_filename}"
 soundfont=$soundfont
 fluidsynth -F "./data/audio/$wav_filename" ${soundfont} "./data/midi/$f"
done
```

Note: the project defaults assume that the `data` folder is inside of the project's `src` folder.

Model Design

I based my model off of a simple but proven example of machine translation using LSTM layers. This seemed like a reasonable approach, as I attempted to model the task as a translation problem.

The network was designed to predict a midi token event E+1 given a sample of audio and event E (which occurs during that audio sample). Using a recurrent approach, each time we predict an output event, we use that as the new input seed (with the same audio sample) and predict the next event in the sequence. This is an approach that works for natural language and can support unequal sequence-to-sequence mapping problems, so it seemed a good approach.

My network would be very simple (only 3 layers - LSTM encoder, LSTM decoder, and Dense output). By comparison, Omnizart's network has over 200 layers. I decided to try keeping it simple as it would make for more productive experimentation and rapid iteration (and avoid overfitting).

Training the Model

Provided your environment is set up and the data has been preprocessed accordingly (and the 'data' folder is inside of the 'src' folder), you can use the `run.sh` script I've provided to train the model:

```
./run.sh train
```

This will run training using the defaults for epochs, early stopping, etc. specified in `main.py`. It also limits the dataset to a single song by default, because a single track can produce over

40,000 audio-midi sample pairs. You'll get a graph of train vs validation loss and accuracy over time, and the network will be saved to disk after training for easy reuse later.

If you'd like to change the training settings, modify the `main` function defaults in `main.py` and rerun the run.sh script:

```
def main(
   num_epochs: int=100,                       # number of epochs to train for
   batch_size: int=1,                         # batch size for training
   latent_dim: int=16,                        # latent dimension for the encoder/decoder
   max_decoder_sequence_length: int=10,       # maximum length of decoder sequences
   song_count_limit:int=1,                    # number of songs to use for training
   shuffle_data: bool=False,                  # shuffle the data before training
   audio_duration: int=10,                    # max length of audio input for prediction
):
```

Using the Model

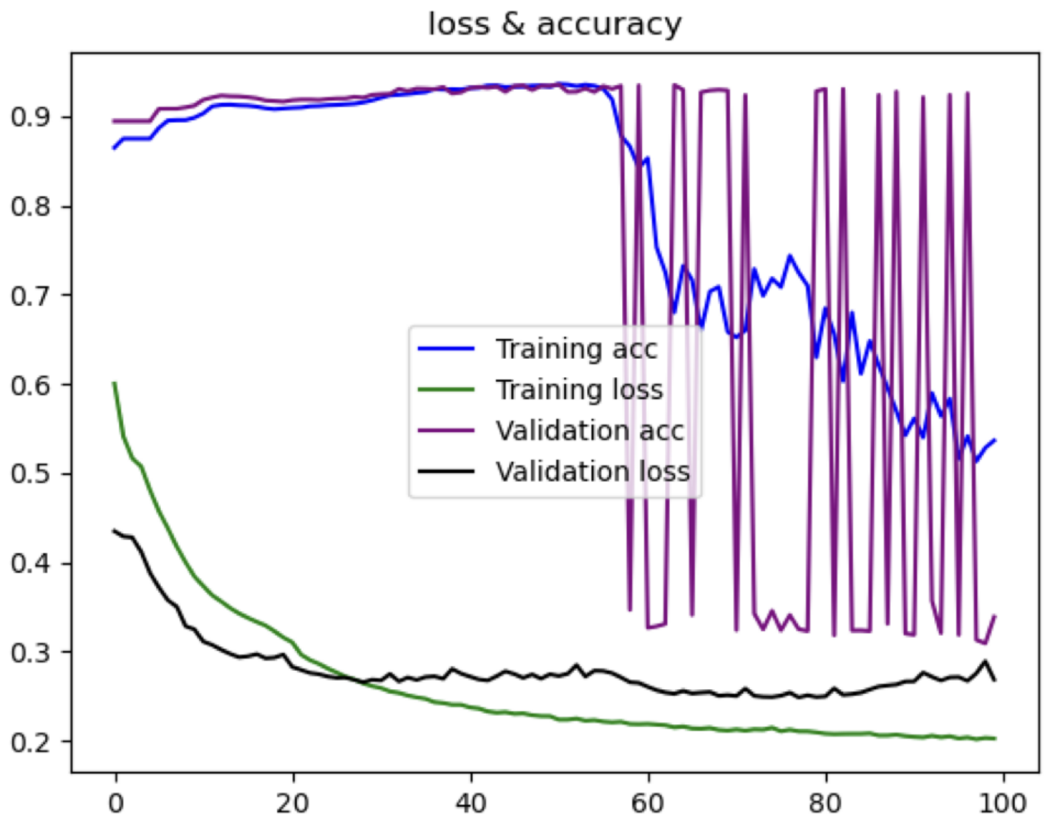
To convert an audio file into midi, use run.sh:

```
./run.sh convert ./data/audio/<filename>.wav output.midi
```

This will load the trained network and create `output.midi` based on the input wav file. Note that, because the performance for prediction is terribly slow, it only attempts to transcribe the first 10 seconds of the audio by default. This can be changed via the `audio_duration` parameter in `main`.

Results

Training on about 45,000 audio-midi sample pairs for 100 epochs, I achieved the following loss/accuracy results:



We can see that the accuracy starts very high (close to 90%) and begins increasing slowly as the loss decreases. However, we hit a threshold around 55 epochs where the accuracy begins dropping dramatically even though the loss continues to decrease. While the results appear surprising, they are actually quite illustrative of the problem space.

The note-seq library used to encode the midi data produces a sequence of midi events, including both Note and Shift events. There are Note Start events which indicate the beginning of a midi note as well as Note End events which indicate the ending. A Shift event represents a slice of space to be placed between other events. So a “note” is encoded as Note Start + many Shift Events + Note End. Here’s an example subsection of midi events:

```
(Pdb) outputs[7]
array([   1,    1,    1,    1,    1,    1,    1,    1,    1,    1,
1300, 1229, 1051,    1,    1,    1,    1,    1,    1,    1,    1,
1, 1, 1300, 1129, 1051])
```

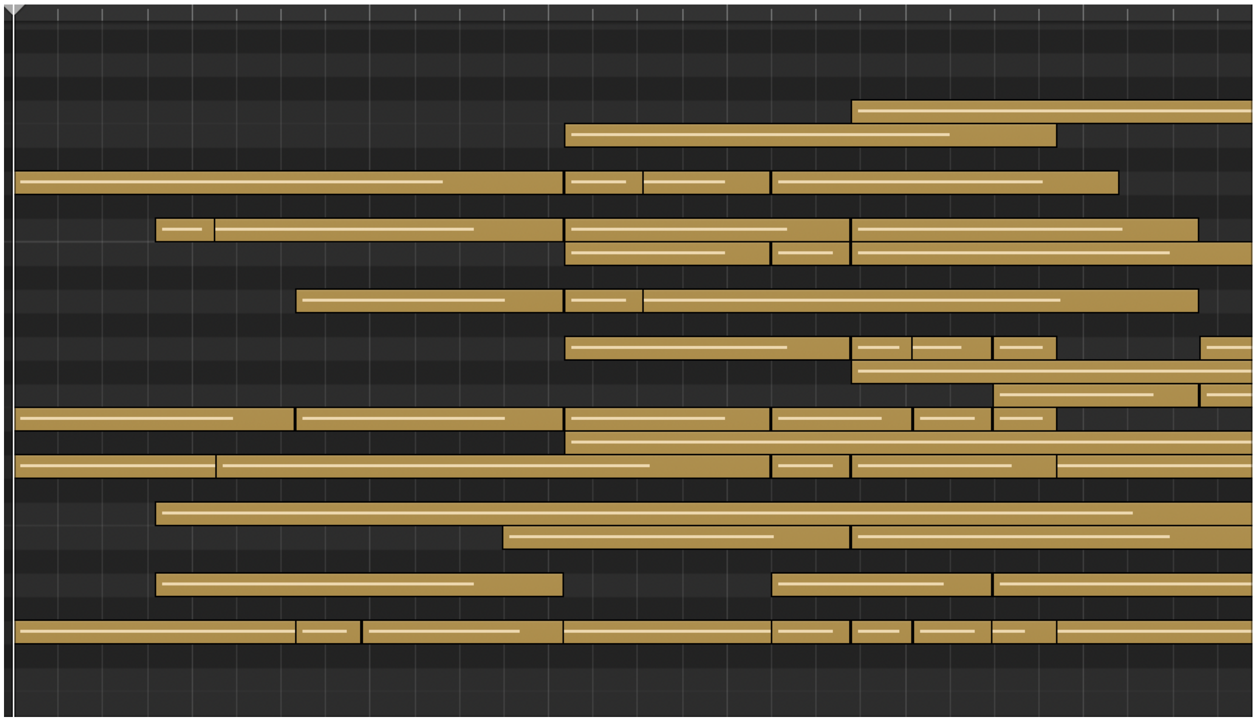
- The 1300, 1229, 1051 subsequence is a “note start” event
- The 1300, 1129, 1051 subsequence is a “note end” event

- The 1 events are all Shifts, which imply the amount of time that note is being played (or the gap between other notes)

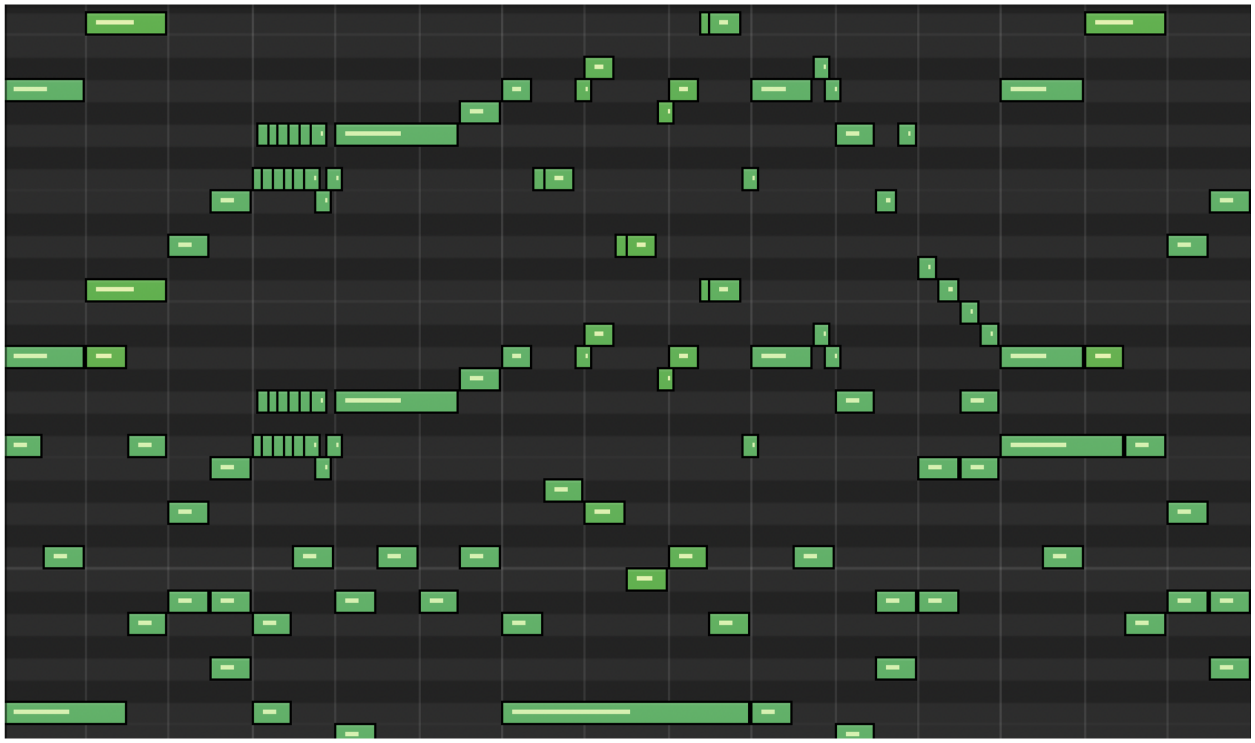
Because of this, for any given midi file, the distribution of events is *heavily* skewed toward Shift events. That means that the network can achieve a great accuracy/loss baseline by simply predicting 1 (“Shift”) for every output regardless of input. This is why the accuracy starts off fairly high (in our example here above, 76% of the events are Shifts).

As the network learns to predict non-Shift events, the accuracy drops as it needs to learn to produce them in exactly the right places. We can explain the oscillation in the test accuracy as the network fluctuating between two weight distributions: one which predicts lots of shifts, and one which predicts lots of non-shifts.

Using this model to then predict midi from a wav input, the resulting midi is not terribly accurate:



The midi it should have produced is:



Note that the color here is also significant as it represents the notes' velocity (which you can think of as volume). It did not perform well on reproducing the expected sequence or note durations, however it did appear to generally learn the musical key of the target audio.

What went wrong / problems encountered

1. A bug in the Magenta MT3 library 😱

The first obstacle I encountered was an apparent bug in the MT3 library for converting midi events back into a midi file (I've filed a Pull Request to fix it). While this caused me some headache initially, I think it is more illustrative of the fact that automatic audio transcription is still a very bleeding-edge domain for research, and while there are some great tools and datasets out there available for use, support for the domain lacks a level of polish (as opposed to topics like e.g. handwriting classification which is now considered fairly trivial).

2. Musicnet labels

As mentioned, part of the reason I regenerated all of the Musicnet wav files was because they didn't match the length of the provided midi counterparts. E.g. a track could be 6:45 long while the corresponding midi track would be 6:20 long. I had considered solving this via uniform time compression/expansion on one or both of the sources, but realized this was a bad idea.

The Musicnet dataset is all *classical* music. One of the hallmarks of classical music is the fact that it "breaths" tempo, meaning it does not follow the beats-per-minute (bpm) tempo exactly. Rather, the tempo of the performance ebbs and flows with the mood of the music (or at the whim and discretion of the conductor). That means that, even though a tempo of 120 beats per minute may be written for a set of measures, in practice they can fluctuate between each measure appreciably (or even within a measure).

The problem is that the provided midi was quantized to a bpm tempo while the performance was not, meaning that no time segment of the midi could reliably be used as a label for the corresponding time segment of the audio.

Note: The Musicnet csv labels supposedly match the recorded audio, however the authors note that there is about a 4% error rate there as well.

3. Input Output Mapping

Once I had a consistent set of input wav and output midi mappings, the next challenge was to construct the network architecture to produce some midi events given a slice of audio input. I realized this was not trivial because I was dealing with *polyphonic* audio; that is, audio which can contain many sources playing notes at the same time. In terms of data, this means that a sequence of 16,000 floats from a .wav sample could correspond to 0 midi events (silence) or several hundred midi events (e.g. a 70-piece orchestra playing Flight of the BumbleBee). If I had instead focused on monophonic audio, the output mapping would be simpler (either there's a note or no note).

In my first attempt to model this problem (before switching to sequential token generation), I sliced the audio and midi into 1 second regions and found the max number of midi events that existed across all regions. I padded all other labels with 0s to match the shape and then constructed my network output vector to produce this shape. However, both the training performance and resulting midi output were quite poor.

I believe the main cause was because I was modeling the problem in such a way that the output could be "fuzzy" - outputting a float for each of N possible positions, scaling back to the range of valid midi events and rounding to integers. Most of the output contained extremely long notes - the network generated the start event, but took a very long time to create the end event. Probably it tried to create an end event several times but for example predicted a number which got rounded to a different type of event after descaling.

4. Data volume[1]

Normally audio is transcoded as 44100 floating point values per second. Even after downsampling to 16000 (still high enough resolution to be intelligible), it still meant many thousands of inputs for each slice of data. I went down to 0.2 second slices for each input, which still meant 3200 individual floats in each input row.

I saw that the MT3 library imported the ddsp library to apply some kind of audio preprocessing called logmel before training their network. I could not easily replicate this because the ddsp library's dependencies don't compile for my computer hardware[2]. I decided not to pursue this because I figured that a sufficiently complex neural network could learn whatever kind of transformation was necessary to map the inputs. While I still believe this is correct, in hindsight I should have spent more time finding a way to preprocess the inputs, as the sheer volume of data made training on more than a few songs at a time too costly for my available computing resources.

5. No 'start' or 'end' token

Though I tried to model the transcription task as a translation problem, there was one key structural difference in the data formats: unlike natural language, a given slice of music has no 'start' or 'end' token to indicate that the subsequence is finished. Since audio to midi is an uneven sequence-to-sequence problem, it's important to know how many midi tokens should be predicted for each slice of wav input. I based my midi reconstruction on an average of 10 tokens per audio slice (hard coded as such) under the assumption that, while we'd lose some events and gain some extra ones, the overall result would still be good. This assumption was totally wrong in practice and I believe it is the largest opportunity for improvement (I could manually craft a designated 'start' and 'end' midi event to use).

6. An Unsolved Domain

The domain of audio transcription is still a not fully solved domain. Google's Magenta MT3, the leading model for the domain, achieved less than 85% accuracy in their report, and my experimentation with Omnizart (another leading framework) also revealed subpar results (much better than mine, but not great either). While both of these leading projects have some very interesting open source tools available (some of which I used for my implementation, like note-seq), these often have dependency problems or suffer from bitrot (for example the Dockerfile for Ominizart fails to build because the Ubuntu version specified is 4 years old).

[1] No pun intended.
[2] I don't know what "pro" is referred to in "M1 Pro", but it certainly isn't "professional data scientist".

Conclusion

While my results did not meet my expectations, I think I reached the practical limit of what I could achieve with the resources available. I learned a lot about the state of research in the music domain and was exposed to many fascinating tools and libraries for music prediction and generation among other things.

I did create the foundation of a network which can produce midi from audio and have some clear paths to improve it: start-end token labeling, increasing network complexity, changing network type, finding better ways to preprocess the audio, and acquiring more computing power to train larger data sets. As a musician and data science enthusiast, I am optimistic for what the future will bring to this area.

References

- Project Code
- YouTube Video Presentation
- Magenta MT3
- Bug in Magenta MT3
- Note-seq
- Omnizart