MIDI Music Generation with LSTM

Team

Danil Meshcherekov - d.meshcherekov@innopolis.university

Project topic

This project aims to create a deep learning model that would be able to generate piano music.

The MAESTRO dataset [1] was used as the main source data. It is a diverse collection of 200 hours of piano performances that were recorded using a MIDI capture system and includes exact time and pressure of each key press and release.

The project involves preprocessing the tracks in the dataset into a format suitable for a NN, designing a NN, and creating an interface for training.

Link to the repository

Link to the github repository with a README file can be found at this link: https://github.com/Arloste/MusicGeneration

What you tried to do during the project

In this project I attempted to create a deep learning model that would be able to capture the underlying patterns and structures of music.

From the beginning I considered it as a time-series problem. So I was mostly inspired by [2], where I got the idea of how to utilize a LSTM using a sliding window and “look back.” During the project I mostly focused on LSTM-based architecture because so far that is the most complex unit I have used and I wanted to experiment more with it.

So, I wasted the entire time trying to come up with some creative ideas on how to predict notes. The biggest problem during that process was that I could only use CPU for training, as I ran out of GPU quota. In the end, I decided to stop on the solution I describe in the next section, but I didn’t have time to even train the model.

Main results (including artifacts)

1. Dataset analysis

The initial dataset comes split between multiple folders by the year of performance; each file follows naming conventions but doesn't contain the composer's name or canonical title of the piece. However, there is a .csv file with metadata (canonical composer's name, piece title etc.):

```
maestro-v3.0.0
|
├── maestro-v3.0.0.csv
|
├── 2004
|    ├ MIDI-Unprocessed_SMF_02_R1_2004_01-05_ORIG_MID--AUDIO_02_R1_2004_05_Track05_wav.midi
|    ├ MIDI-Unprocessed_SMF_02_R1_2004_01-05_ORIG_MID--AUDIO_02_R1_2004_06_Track06_wav.midi
|    ├ MIDI-Unprocessed_SMF_02_R1_2004_01-05_ORIG_MID--AUDIO_02_R1_2004_08_Track08_wav.midi
|    ...
|
├── 2006
|    ├ MIDI-Unprocessed_01_R1_2006_01-09_ORIG_MID--AUDIO_01_R1_2006_01_Track01_wav.midi
|    ...
|
...
```

Each of the MIDI files follows a structure: it sets the same speed and tempo for all files and stores the music information in track 1. Also, it doesn't store "note_off" messages (that correspond to releasing keys), but uses a "note_on" with zero velocity. Also, it includes messages about sostenuto pedal control. That is a sample information in a midi track:

```
Message('control_change', channel=0, control=64, value=119, time=19),
Message('control_change', channel=0, control=64, value=124, time=49),
Message('note_on', channel=0, note=27, velocity=34, time=758),
Message('note_on', channel=0, note=39, velocity=36, time=1),
Message('note_on', channel=0, note=27, velocity=0, time=62),
Message('note_on', channel=0, note=39, velocity=0, time=40),
Message('note_on', channel=0, note=46, velocity=42, time=410),
Message('note_on', channel=0, note=46, velocity=0, time=338),
...
```

2. Dataset representation

The most convenient way to store the data is by using three arrays of the same length for storing notes, its delta time, and velocity separately, where each value corresponds to a note. For the sample above, that is how I store the values:

```
notes = [27, 39, 27, 39, 46, 46]
times = [758, 1, 62, 40, 410, 338]
vlcty = [34, 36, 0, 0, 42, 0]
```

Additionally, I use augmentations on these values (although not when pre-processing dataset, but when training the model for each sample separately to save space). Augmentations involve transposing the notes (adding a constant integer to each note value), stretching the time deltas, and changing the loudness (velocity).

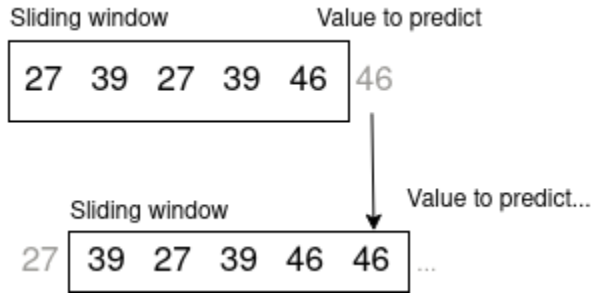
Internally, I store the dataset with initial (but renamed) MIDI files and a torch data loader for each model that implements the functionality above.

3. Approach to the problem.

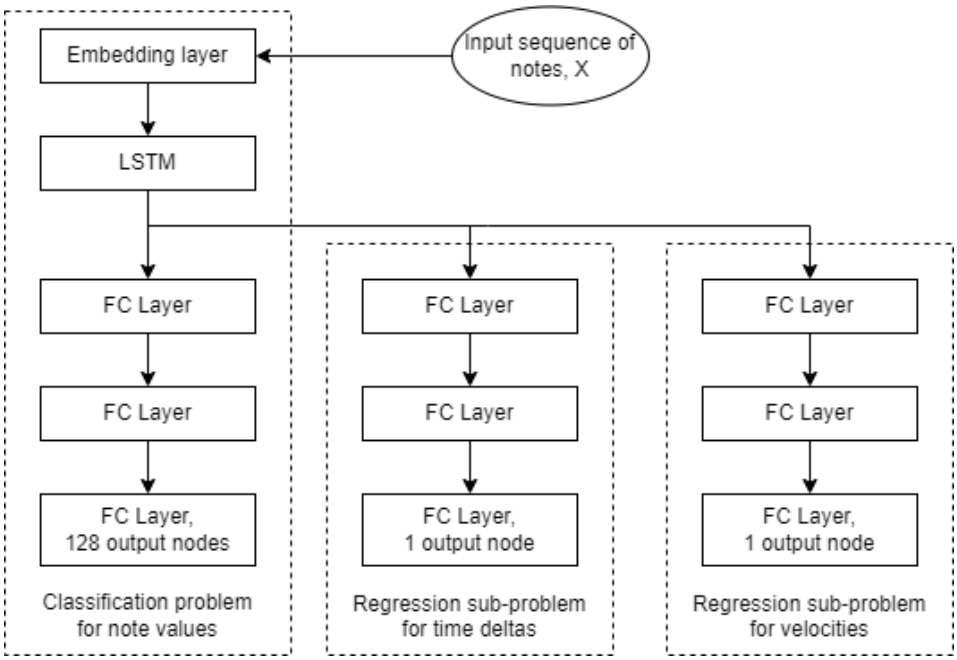
Right now we have three series that together represent some music. Therefore, it is quite reasonable to treat this problem as a time-series problem.

However, it turns out that building three independent LSTM units for each series or trying to feed all of them into one LSTM is impractical. So I decided to treat only the note-series as a time-series, because it is the most important part, and predict time deltas and velocities as regression sub-problems for each predicted time-series value.

The architecture uses a sliding window of notes. This is an example for the sample above:



This is the diagram of my proposed architecture:



As one can see from this diagram, I split the problem of predicting three parallel time-series into one time-series problem, and two additional subproblems.

Firstly, I feed the entire embedded input sequence to the LSTM layer, and then run three parallel pipelines consisting of fully-connected layers. The first pipeline tries to solve a classification problem and has 128 output nodes, which correspond to each of 128 possible notes. There is a separate optimizer that only optimizes the parameters related to the time series and classification.

The other two subproblems are the task of associating each item in the time series with a numerical value. Given the previous notes (or, more precisely, given the LSTM output), it should predict these numerical values of the next item in the time series. Each of these two subproblems have their own optimizer that only affects the parameters in corresponding pipelines, without affecting the embedding and LSTM layers.

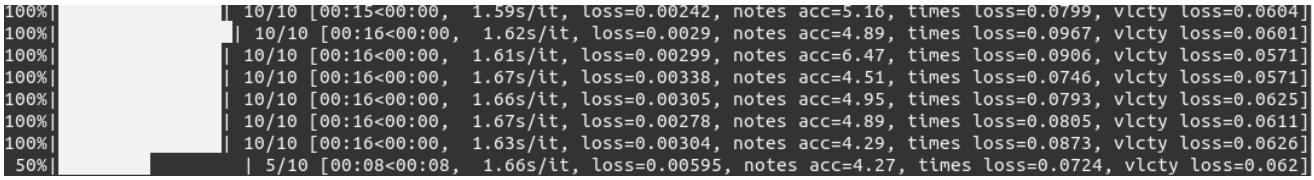
4. Music generation

When training the model, I also used empty input without notes (a list of zeros) to teach it how to start music. As such, when generating a new piece, it is first fed an empty sequence, then the output is used as input, etc. No prompt is required; but the results may differ due to non-deterministic dropout layers.

5. Final result

The biggest result I achieved is the program that is able to train models and use them to generate MIDI files. It is possible to do by opening the terminal in the root directory of the project repository and running the main.py file. You will be prompted to either create and train a new model, continue training an existing model, or generate a MIDI file using a model.

However, I do not have any tangible results because I failed to have trained the model, and the current output is not satisfactory. Right now, as I am writing this, the model that is trying to replicate Debussy's pieces is training in the background, and I'm not getting any coherent results from it: the note, time, and velocity prediction accuracies are far from being good :(



Timeline of the project

In the beginning of the project my initial idea was to generate .wav files using multihead attention and style transfer to be able to generate coherent and lengthy music files with specific mood, style of one's choice. None of those ideas were realized, however.

After a couple of weeks of testing different configurations and ways to compress wav files I gave up the idea of generating wav files as it turned out to be impossible to train a wav model using only cpu, and google colab GPU quota is limited.

Also, I resorted to using a simpler LSTM-based NN architecture. Transformers would be also very difficult to use as I have no experience with them, vanilla transformers do not show good performance with music [2] and also not suitable for for working with MIDI as each note in a MIDI file has one numerical and two categorical features, as opposed to only one "channel" in texts or wav files.

Later, although style transfer looked simple enough - it was just a matter of using pre-trained layers for predicting next notes and their length and dynamics - but I abandoned this idea too because I was unable to fine tune the model to perform regularly, let alone adapting styles for it.

I spent another couple of weeks just thinking about possible improvements and trying to fine-tune the model. I discussed the current version above. Due to my inability to manage time, I only released the general framework for training the model, but I could not train an actual model.

Individual contribution of the teammates

As I am the only member of my group, everything in this project was done by me.

References

[1] https://magenta.tensorflow.org/datasets/maestro

[2] J. Brownlee, “Time Series Prediction with LSTM Recurrent Neural Networks in Python with Keras.” Accessible at https://machinelearningmastery.com/time-series-prediction-lstm-recurrent-neural-networks-python-keras/

[3] A. Cheng-Zhi *et al.* “Music Transformers.” Accessible at https://arxiv.org/abs/1809.04281