Lab 4 - Music generation with RNNs

Vidaich Francesco

1 Introduction

The goal of this assignment is to build a model that is able to generate music given an initial sequence. To train the network we used the piano-midi dataset, which is composed of classical music in the form of MIDI files. The PyPianoRoll package was used to handle this format and convert the data to numpy arrays. The main component of the NN is the LSTM layer, but different architecture were tested: here we will discuss about the most complete one (called *two-hands model*), but the simpler *original model* was kept as an older version.

2 PyPianoRoll package

PyPianoRoll is a python package for handling multi-track MIDI pianorolls and convert them into numpy arrays with 3-dim shape (indexes are *track_id*, *time* and *pitch*). Each pitch is represented by a number. In particular, since every octave is composed of 12 different pitches, their beginnings (note C or Do) correspond to a multiple of 12. For example, the zero of the axis corresponds to C_{-2}, the Middle C (i.e. C_4) of a grand piano is represented by the number 72 and the highest pitch is defined by the number 127.

3 The *piano-midi* dataset

Although the *piano-midi* was not the biggest dataset, it was the most carefully built between all the available ones: many of the operations used to pre-process the data would not have been possible with other datasets. In particular, the timestep of its MIDI songs is not fixed to a time interval of few milliseconds, but it is equal to a small fraction of a beat. This means that it changes as a function of the tempo of the song and that a beat from any song will be always represented by the same number of timesteps. Moreover, its MIDI songs contains always two tracks, one for the notes played from the right hand and the other for the left hand. This made possible the development of the *two-hands model*.

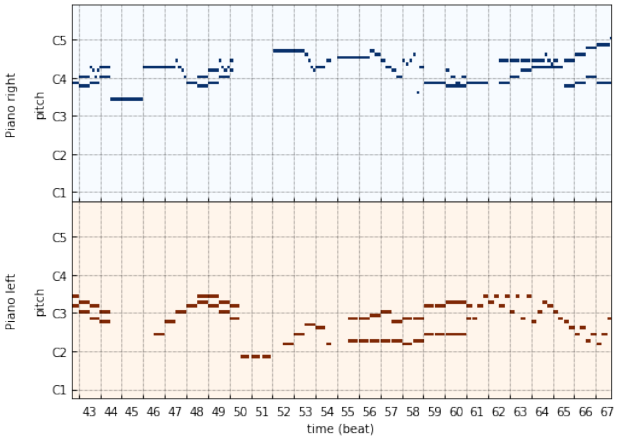

Figure 1: *Portion of a MIDI song from the dataset visualized with PyPianoRoll*

3.1 Pre-Processing

In Python, the MIDI acquisition and conversion is really slow, so it is helpful to store all the songs as numpy objects before building the training dataset. This is done by the script called *numpy_pre-process_script_two_hands.py*, which performs also some operations needed to reduce the complexity of the data:

- Volume intensities of all notes have been deleted in order to represent MIDI files as boolean numpy matrices;
- Since the grand piano keys do not cover all the 128 possible pitches encoded by the MIDI files, we can crop their pitch range and consider only the ones in its correspondent interval $[21, 108]$.

- MIDI files of the *piano-midi* dataset have a time resolution of $(1/96)$th of a measure. By sampling one timestep of the song every 6 of them we obtain a resolution of $(1/16)$th, which is often the highest used by musicians (we are not interested in generating music with higher resolutions)

3.2 The Dataset Class

In order to organize and load batches of sequences for training, we used the `Dataset` and `Dataloader` classes provided by `torch.utils.data`. The dataset class is called `Piano_Dataset_two_hands`. Since the model will be trained with sequences of a fixed `sample_length`, we aim to extract all possible ones from the dataset. The main class member is `self.all_songs` and it is a dictionary that contains the preprocessed numpy arrays of all songs. From each song we can extract `(song_length - (sample_length+1))` sequences. Each sequence of the dataset is identified by a tuple containing its starting point and the song ID, and every tuple is stored in the `self.indexes` member of the class.
Since the piano-midi dataset is a bit small for training our network, we choosed to increase its size by storing also transposed version of the same songs. This is done by just shifting the notes along the pitch axis (if a song exceeds the piano range of $[21, 108]$, it is discarded). For example, if `tunes = [-2,-1,0,+1,+2]`, the new dataset is almost 5 times bigger than the original.
To store more transposition of the dataset in memory, we also added the `encode_sparse_mat` argument for the initialization of this class. If this flag is true, the sparse matrices describing each song are encoded to two vectors of arrays of indexes (corresponding to the notes played at each timestep for each hand). The value of this flag is stored as a member of the class since we need to decode each sequence when running the `__get_item__()` method if the flag was true. This method returns the requested sample, the notes played by each hands in the following timestep and the number of notes played by each hand in that timestep (see next section).

4 RNN Structure

The network class is called `RNN_two_hands_net`. It is based on a LSTM layer that takes in input the 88 bool variables (keys of the piano) of the right hand and another 88 binary variables for the left hand at each timestep (the input layer has 176 nodes in total). The LSTM layer takes in input also its state after the previous timestep to understand the temporal structure of the sequence in input. In our case, the LSTM had 2 hidden layers of 150 nodes each. After this layer, the network splits into two branches: one has to forecast which notes will be played by the right hand and the other has to do the same job for the left hand. After some dense layer in each branch, both outputs layers are given by an array of 88 probabilities for every notes to be present in the timestep following the input sample and by a variable N_R (or N_L) that represents the number of notes that the hand will have to play. Ideally, the following timestep can be generated by sampling N_R notes from the right distribution and N_L notes from the left one, but some adjustments were needed to improve the results (see section 6).

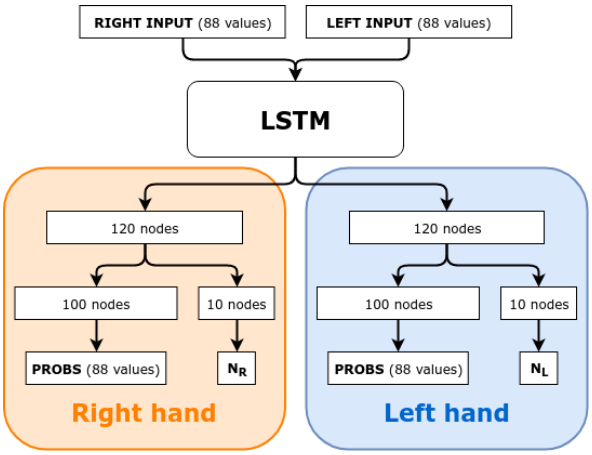

Figure 2: *Simplified scheme of the two-hands architecture*

The activation function used before each hidden layer is a `leaky_relu()` and all output layers have a linear activation. For both hands, the loss function for the number of notes is given by the MSE and the one used for evaluating the 88 piano probabilities is `nn.BCEWithLogitsLoss()`, which combines a `nn.Sigmoid()` layer and the `nn.BCELoss()` in one single class. The `nn.CrossEntropyLoss()` could not have been used since each hand can play multiple notes at each timestep.

5 Training the model

The function that executes a training step is called `train_batch()` and it is a method of the network class. It takes as arguments all the objects needed to perform the training, like the batch,

loss functions and the optimizer. It is possible to notice that there are two different loss functions to evaluate right and left piano probabilities: each loss function is initialized with a correspondent `positive_weights` argument. This is an array of 88 values that represent the ratios between the number of timesteps in which a note is not played and the ones in which it is; it is computed by the `get_pos_weights()` method of the `Piano_Dataset_two_hands` class in order to give more importance, when computing the loss values, to notes that are less frequently played in the songs. The final loss value that will be used for backpropagation is obtained combining the single losses in the following way:

$$L_{final} = L_{right} + L_{left}$$

where each term is equal to

$$L_{hand} = \alpha L_{piano} + L_{num\_notes}$$

In the code, the constant α is called `loss_coeff` and balances the impacts that the different types of errors have during training. By default the L_{piano} is greater than L_{num_notes}, but `loss_coeff` was still set to 1.2 because predicting which notes are more plausible after a sample is a much harder and important task than finding how many notes to play.

The model was trained over samples with `sample_length=32` timesteps (which corresponds to 8 beats of music), grouped into batches of 100 samples. Using `tunes = [-2,-1,0,+1,+2,+3]` we built a dataset 6 times bigger than the original (maximum size that could have been stored in memory), so few epochs were necessary to complete the training. We used the Adam optimizer with `lr = 0.001` and a `weight_decay = 0.00001`. Performing the training with these parameters and computing the mean value of each loss after every 2000 batches, we obtained this evolution:

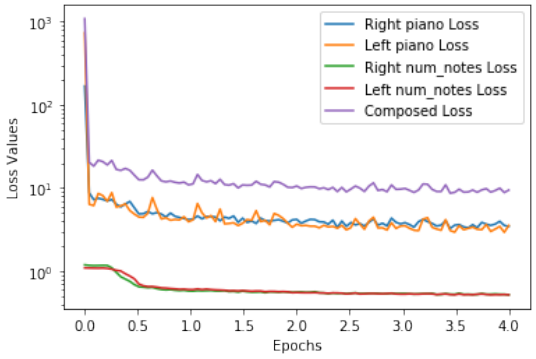

Figure 3: *Loss values evolution during the two-hands model training*

This training process was performed by a GPU and therefore took only about three hours to finish. After that, the trained model and the loss values shown in Figure 3 were stored in the *trained_models* folder. In Figure 4 and 5 it is also shown an output example of the model for both right and left hands:

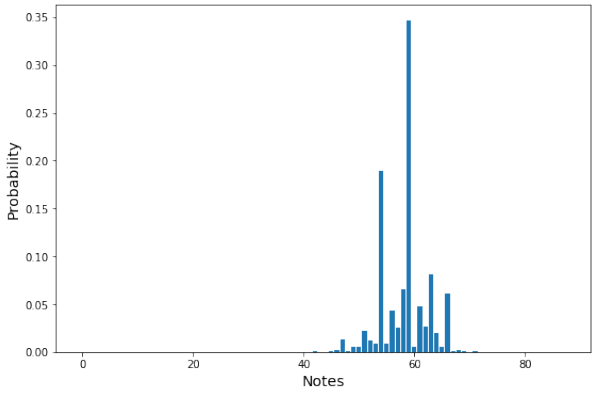

Figure 4: *Probability distribution for the Right hand, in this case we had $N_R = 1$, so one note had to be sampled*

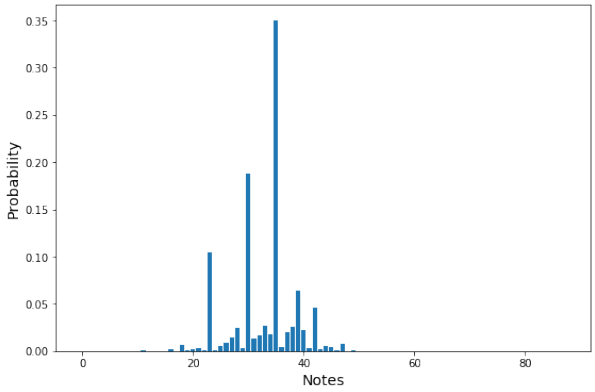

Figure 5: *Probability distribution for the Left hand, in this case we had $N_L = 1$, so one note had to be sampled*

6 Sample Generation

The script *generate_two_hands_sample.py* loads the trained model, takes the first 32 timesteps (8 beats of music) of one song of the dataset and continues it by generating an arbitrary number of following timesteps. The model takes in input the loaded sample, updating the `rnn_state` after each analyzed timestep, and then produces the following sample in the desired way. At the beginning of the code there are some parameters that can be changed: some defines the names of directories and files needed to produce a sample, others can tweak how this

sample can be generated. One of the most important flags is called `sample_RL_notes` (one for each hand): if this is true the notes are sampled from the distribution estimated by the model, if not the selected notes are just the ones with highest probability. Both cases had their drawbacks: the first produced samples where notes change after almost each timestep (there many "good ways" to continue a melody, so they have the same probability to be picked) making it sound a bit random, and the second produced samples where the model chooses a "good" chord and kept it fixed throughout the whole generated sample.
To solve this problem, we introduced the `apply_probs_corrections` flag, which increase the probability of notes played in the previous timesteps if they are sampled and decrease them if the notes with highest probabilities are the ones selected. The corrections are defined as

$$p_{sample} = (1 + be^{-an})p_{orig} \quad \text{and} \quad p_{max} = e^{-an}p_{orig}$$

where $a \in [0, 1]$, $b > 0$ and n is the number of times that a note is consecutively played (in both cases, if the note is not played in the last timestep the correction is not applied). The second method continued to have problems because the model kept the same chord for a while, changed it for one timestep to reset the penalization and then returned to the same chord (check the *penalty_cant_work.mid* result to see that). The "sample" method improved from this correction and therefore it is recommended. At the moment, the chosen parameters are $b = 1$, $a = 0.1$ for left hand and $a = 0.25$ for right hand (because it has to build the lead melody, thus it should change notes more often), but new values could be tested. At the end, the generated sample is stored in the *generated_tracks* folder as a MIDI file and it is printed in a plot similar to the one shown in Figure 1.

7 Conclusions

To obtain the current results, a lot of tests were done, and each improvement was made to try to fix some problem of the older version of the model. Despite this, the quality of the generated samples has not reached the desired levels because it was not able to handle any 'high-level' structure of a song. We tried to increase the `sample_length`, but this caused even worse results. This could be due to the complexity of the output that the network has to generate: other architectures works better with encoded inputs and outputs, but they could produce only melodies or simple chords at each timestep. Since we wanted to give the model more freedom, we avoided this solution and started developing the explained architecture. It is possible to try combining the two strategies (like keeping the two-hands architecture but working with simple encoded data), so hopefully further studies could bring better results.