Classical 846 LSTM

Aaron A. Gauthier, Avijeet Kartikay and Pedro Uria Rodriguez

Machine Learning II

The George Washington University

Classical 846 LSTM

Introduction

Background. In the 6th century AD, Pythagoras proposed a concept called *The Music of the Spheres* to describe the proportional movements of the celestial bodies in the sky. However, this was not music as we think of it today, but rather a mathematical representation. Math and Music are intrinsically intertwined. The field of algorithmic composition dates back to the early days of Computer Science. Hiller and Isaacson at the University of Illinois Urbana-Champaign were the first to use computers to create music. They programmed the university's Illiac I computer system to generate music tones based on random numbers. They utilized a model based on Markov Chains to predict the future path (probabilities) of a music note. If the note was not a fit, a new note was generated. The score that was generated in 1957 was called the Iliac suite string quartet. However, the music community did not immediately accept this as a true form of music, even though the scientific journals accepted and published their work. It was not until after Hillers death that their work was accepted and published within the music community.

Current Situation. Today, generative programs such as iTunes amongst others are commonplace to aid humans in composing music. There are 3 different types of models used for the composition of music. The first is a translation model. Translation models are rule-based models that take a non-musical medium like a picture and translate it into a rule-based algorithm for music. An example would be if it sees a horizontal line in a picture the rule may translate it as a constant pitch, whereas if it sees a vertical line it may translate it as a moving pitch or extended scale. The second is based on genetic algorithms such as mutation and natural selection. Different compositions essentially evolve into a suitable composition. The third model is the learning model, which is what we will focus on. In particular, we plan to use a LSTM network to learn the underlying probability distribution of the musical pieces and generate new music based on it.

Data

We will be using Musical Instrument Digital Interface (MIDI) files. MIDI is basically a musical alphabet. You can think of it as a sequence of events, where for each event there is information about the note/rest/chord being played and its duration, together with the tempo, the intensity and other musical metadata. It supports multiple tracks and instruments. Our main source of data came from the Classical Piano Midi Page, which contains many classical pieces organized by author and genre. This was very good news for us because we wanted to train our networks on somewhat similar music, at least at first. Later on we could build up and mix different authors, but we have not done so for this project. We also have other

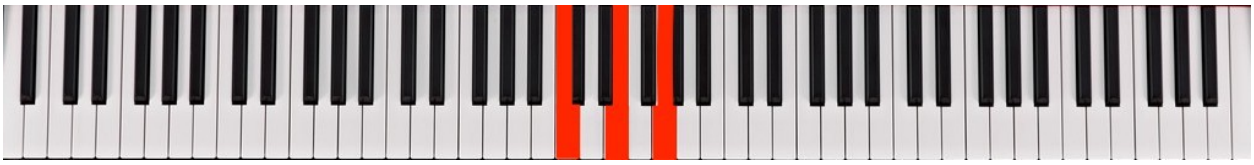
sources of data, but it is worth noting that

"... The quality of the pieces, however, may vary considerably"

as stated by the owner of Classical Piano Midi Page. This is in a big part due to the existence of audio to MIDI converters, which do not produce good quality files. These softwares are actually trying to solve a very hard problem called *audio source separation*, and no one has completely yet succeed.

Encoding

Different components. In order for our neural network to process music, we need to encode this music into a sequence of vectors with numerical values and of the same constant temporal duration (*time step*) . These values need to represent the musical info of each time event (each note being played at a certain time). Each time event contains an `offset`, which indicates its temporal location. This will correspond with the position of the vector on the sequence. There is also the pitch, i.e, the frequency of the note/s being played. As we will be focusing mostly on piano music, we are considering 88 different notes. Thus, we took a *many-hot-encoding* approach, in which we create a 88 dimensional vector with 1s on the position of the notes being played, as shown below:



$$\bar{p}_t = [0, 0, ..., 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, ..., 0]$$

We also need to account for rests, so we create an extra dimension to do so. Finally, there is also the duration of each time event. If a note is longer than the *time step*, it will be split into more than one vector. Suppose we have *time step* = ♩ and event at t_i = 𝅗𝅥. Then we decode it as $\bar{p}_{t_i} = [0, 0, ..., 1, 0, 0, ...]$ and $\bar{p}_{t_{i+1}} = [0, 0, ..., 1, 0, 0, ...] = \bar{p}_{t_i}$, so we actually have ♩ ♩, instead of 𝅗𝅥 , which is obviously not right. To avoid this, we add another extra dimension called *hold*. Then, our encoded vectors will be $\bar{p}_{t_i} = [0, 0, ..., 1, 0, 0, ..., 1]$ and $\bar{p}_{t_{i+1}} = [0, 0, ..., 1, 0, 0, ..., 0]$. The last 1 on $\bar{p}_{t_i}$ indicates that the notes being played at t_i are being held. Thus, we can have successfully encoded the duration of the original note t_i = 𝅗𝅥. Originally, we also encoded the tempo as one extra dimension with many possible integer values, but we realized that there was no way for use to train the network without actually *one-hot encoding* this dimension into many more dimensions, so we deemed it not worth it. The same applies to the volume of each note.

Melody. To generate melodies we only take the right hand of each song as input. Thus, the approach described above completely covers this need. Note that there can be more than one note being

played at the same time, that is why this approach is called *many-hot encoding*. It is also worth mentioning that we forgot to encode the note B_0 (the third note from the left). However, this does not affect our results because notes so close to the left edge are rarely played, and actually, we could have even encoded less notes and it should have been fine. This might had actually make it easier for our networks to learn.

Melody & ***Harmony.*** In this case things got a bit more complex. As we did not want to increase the number of dimensions, we first thought about combining them together just by adding up the vectors for the left and the right hands. However, with this approach we lose info: the hold dimension on the vectors for the right and left hands gets mixed up and as a result the offsets get mixed up too. We believe there is no way to fix this problem if we decide to encode this way. Finally, we decided to stack them up horizontally, effectively doubling the number of dimensions. So we end up with a vector like

$$\bar{p}_{t_i} = \overbrace{[0, 0, ..., 1, 0, 0}^{\text{left hand}}|\overbrace{0, ..., 0, 0, 1, 0, ...]}^{\text{right hand}}$$

This encoding method, called *multi-many-hot encoding*, will certainly make it harder for the network to learn, but with enough neurons and layers it might still be possible to learn both hands together.

Network Architecture

Overview. As we have mentioned, we used an LSTM to generate music. This was also the reason we encoded the music as a sequence of notes, including the hold dimension. The network consists of an arbitrary number of LTSMs with an arbitrary number of neurons as the Hidden layers, and also with an arbitrary dropout probability on each layer, followed by a fully connected layer with the same number of neurons as input dimensions as the Output Layer. We need this exact number of neurons because we want to predict the next note after a sequence of input notes, thus the output dimensions must be equal to the input dimensions. These will be 89 when generating melodies, and 178 when generating melodies and harmonies at the same time.

Output Layer. Because we are using a *multi-many-hot encoding* approach, we actually have a *multi-label* classification problem. That is, each output vector can belong to more than one class, as we have the hold dimension and also more than one note can be played at the same time. Then, we cannot use the regular *softmax* transfer function on the Output Layer, because this would only classify the output as a single note, as it models the joint probability of each dimension of the output *multi-many-hot encoded* vector. Indeed, this would result in only one dimension having a probability close to 1, while the rest are close to 0. Instead, we use a sigmoid function, which will model the probability of each dimension of our output vector individually, thus allowing for multiple probabilities close to 1, indicating that multiple notes and/or notes of larger duration are being generated.

Performance Index. The loss function used is then also different than *Categorical Cross-Entropy*. We are using *Binary Cross-Entropy*, because we are treating each dimension individually, so we only have two classes for each of them (0 or 1) but our output vector can have many class labels at the same time (many 1s). Our optimizer of choice is `Adam`.

Training

On a single song. After designing the architecture, we had to think about the training process. First, we wrote our code to train the network on only one note, in order to evaluate its ability to learn the pattern within the sequence of notes in such song. To do so, we choose a sequence of notes to input our network of arbitrary length, and then we compute the loss between the predicted output and the real one, that is, the next sequence of notes. So if we input the sequence $[0, 1, 2, 3]$, where each integer represents a note and its temporal position, then we compute the loss between the output and $[1, 2, 3, 4]$. We also wrote another approach where the loss is only computed between $[4]$ and the last note of the output sequence. After computing this loss, we go backwards using the BPTT algorithm, and update the weights and biases. Then we do the same thing with $[1, 2, 3, 4]$ as input and $[2, 3, 4, 5]$ as target, inputing also the hidden and cell states from the last iteration to the LSTM. We repeat this process over many epochs. We also added a condition on the code that decreased the learning rate by half when the loss does not decrease for 10 epochs, as we found this to be a somewhat common occurrence.

On many songs. After making sure our network was able to reproduce a single song with a high degree of accuracy, which was done by using a generator function that will be explained later, we expanded our code to work on many songs. Our first approach was to compute the aggregated loss over all sequences of all songs for a given sequence index. So for example, we have song A with sequences $[1_A, 2_A, 3_A, 4_A, ...]$ and song B with sequences $[1_B, 2_B, 3_B, 4_B, ...]$. If we choose a sequence length of 4, on the first iteration (inside the first epoch), we go forward using $[1_A, 2_A, 3_A, 4_A]$, compute the loss, and then go forward using $[1_B, 2_B, 3_B, 4_B]$ and compute the loss and add it to the previous loss. Only then we go backwards and update the network parameters. We repeat this using many sequences and over many epochs. We split this approach into two sub-approaches. One the first, we input to the network that will take $[1_B, 2_B, 3_B, 4_B]$ as input the output hidden and cell states from the network that took $[1_A, 2_A, 3_A, 4_A]$ as input. On the other approach, we keep track of each different hidden and cell state for each song and only output the corresponding on the corresponding iterations.

Our definitive approach consists on the following: We simply stack all the encoded notes of all the songs on the training data together, sequentially. We end up with a huge matrix of sequences of notes,

which does not distinguish where one song ends and where another song begins. Then we just input this matrix in the same way we did on our approach for only one song. Because looping over the whole songs takes a long time when we use a lot of them, we add the option of looping over only the first n sequences of each song. This way the network will not see the full songs, but will still learn some patterns, possibly more than enough, given that usually, the songs contain repeating patterns along their body of notes. Because we are also using many songs, our network will still see many different patterns.

Generator

Basic concept. Once we have trained our LSTM on some data, the final step is to use this network to generate new music. In order to do so, we used two slightly different approaches. The first one consists on taking as input a sequence of notes sampled from one of the songs we used to train the network, with the same length as the sequence length used to train the network. The rest of the process is analogous to the one described for the second approach below.

The second approach only takes a single note [1] as initial input. Then, we go forward and get a new note [2] as output. This note is actually a vector of probabilities, so we convert it to our multi-many-hot encoded vector of 1s and 0s by selecting either all the dimensions with probability greater than an arbitrary threshold, or if all the dimensions have probabilities less than this threshold, we select the dimension with the highest probability. In the case of generating melody + harmony, we do this separately for each hand. We also add another threshold to include the hold when the probability of such hold is close to the highest probability. After deciding on the encoded note, we add this note to the sequence of notes we will be using as input, so now we have $[1, 2]$, and go forward again to generate, in this case, a sequence of two notes $[2', 3]$. This time we only take the last generated note [3] and add it to the next input sequence $[1, 2, 3]$. We do this again and again, until we reach a sequence length equal to the one our network had been trained with. When we reach this length, say we have $[1, 2, 3, ..., i]$, we use this sequence as input, but when we get our next generated note $[j]$, we remove the first note from our input sequence and add this new latest note, so on the next iteration we will input $[2, 3, ..., i, j]$. We repeat this until the number of wanted generated steps has been reached. Note that each output note (last note of each output sequence) is being added to a list of generated notes even when being removed from our input sequence at some point, and that we can keep going on forever and generate songs of any duration. We also remove the first generated notes until the constant sequence length has been reached, because these notes will not be representative of a network which was trained with a larger length of sequences.

Improvements. When we first trained our networks with the first two training approaches, we run into some issues. One of them was that the network sometimes got stuck repeating a particular note over and and again. To solve this, we added an option (default to `False`) to change remove the last note from the generated sequence when the exact same note was generated 6 times in a row, and add a new note randomly sampled from one of the songs of the training set. This fixes the issue in most cases, although we found that it was not needed most of the time when we were using the definitive training approach. The same thing applies for the network getting stuck repeating a particular sequence. We again added a new option (default to `False`), to fix it. This time it was not that easy because identifying that a sequence of arbitrary length is being repeated is not trivial, so we ended up using a brute force approach in which we just change the note when the number of generated notes is a multiple of the sequence length. Again, this was not needed on most of the music generated by the definitive training approach. Finally, sometimes we also encountered an issue where the music was good but there were too many rests in between the notes. To solve this, we added an option (default to `False`) to remove any rests that have a duration of four times the time step, or if the rests have been repeated during 4 time steps (even if they have no hold on them). These options were mostly needed for our first two training approaches, but for the last we barely used them.

Experiments and Results

Hyperparameters. Our code contains many parameters and options to play with. Below we provide a brief description and some results trying different values for some of them.

1. `hidden_size`: After experimenting with various values such as 356, 712 and 178, the best music was produced with hidden_size of 178 for melody & harmony and 89 for only melody.

2. `seq_len`: A sequence size of 100 works best for songs of Tschaikovsky, Albeniz, while the network generates soothing music for Haydn and Mendelssohn songs with a sequence length of 50.

3. `time_step`: It is the minimum time interval between two incremental events. Initially 0.05 was chosen, but due to the huge computational size of the data, 0.25 was chosen. This reduces the dataset size drastically without loss of the musical quality. A *time step* lower than 0.25 (a 16th note) is usually a very bad idea.

4. `lr`: A learning rate of 0.01 is chosen and this value decays when the loss stops decreasing for the last 10 epochs. The default 0.01 was alright in most cases, mainly thanks to this feature.

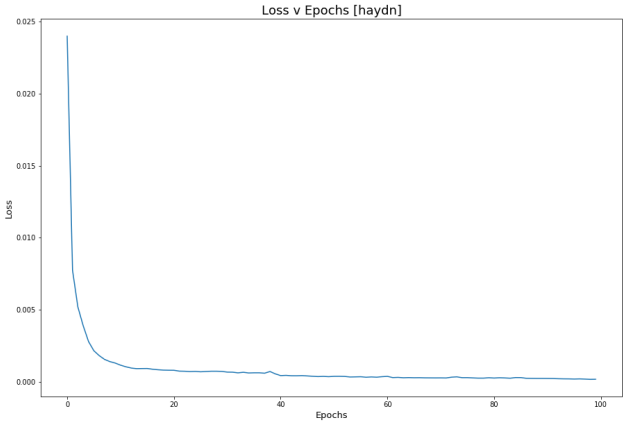
5. `n_epoch`: Roughly 50 - 100 epochs work well with the chosen decaying learning rate. Once a satisfactory loss values is reached, using a keyboard interrupt the training can be manually stopped

and then it can be run again with this many epochs, instead of having to wait the initial chosen number of epochs.

Best Results. Considering the parameters, the best results were seen, or rather heard, by using the second training approach and more heavenly the third training approach, with the following values of the hyperparameters:

1. `hidden_size`: 178
2. `seq_len`: 50
3. `time_step`: 0.25
4. `lr`: 0.01
5. `n_epoch`: 50 - 100

This is an example of the loss obtained over each epoch when training on multiple songs from Haydn:



Conclusions

We have designed and trained a LSTM model, with fairly satisfactory results. We hope to find ways to capture and encode more musical features available to us through the MIDI files, such as, tempo, intensity, etc. Though complex, we aim to extend this idea to genres beyond classical, such as Pop, Rock, Jazz and so on. Other ways to encode the musical events of the MIDI files will also be explored. It can be noted that the problem of encoding the data is analogous to the challenge of encoding messages while transferring them over a connectionless or fully connected network, without any loss of information.

We have seen that the training and generation process for networks trained on different data can be quite different, so there is no real way of getting good results every time. We have had good success on generating melodies, but also on generating melodies & harmonies combined, although the latter were a bit more hectic (as it was initially expected). One approach we considered but did not take due to a lack of time was using our system to generate melodies and then using a MLP/CNN to map these melodies with their corresponding harmonies, using the music we used to train the LSTM as training data, with input being the melody and target being the harmonies. We will consider this in a future approach.

References

[1] Jean-Pierre Briot, Gaëtan Hadjeres, FranÐois-David Pachet, *Deep Learning Techniques for Music Generation - A Survey*, 23 Mar 2019 . https://arxiv.org/abs/1709.01620

[2] Martin T. Hagan, Howard B. Demuth, Mark Hudson Beale, Orlando De Jesús, *Neural Network Design* http://hagan.okstate.edu/NNDesign.pdf

[3] https://web.mit.edu/music21/doc/

[4] https://pytorch.org/docs/stable/index.html

[5] https://gombru.github.io/2018/05/23/cross_entropy_loss/

[6] https://www.depends-on-the-definition.com/guide-to-multi-label-classification-with-neural-networks/

[7] Classical Piano Midi Page