By: Marwan Harajli
Subject: Springboard Capstone 2 Second Milestone Report
Last Updated: 04/06/2019
Project Location: https://github.com/harajlim/MusicGeneration

Introduction:

In this report we present progress so far in what relates to the Springboard second capstone project. For my project, I decided to look into music generation using machine learning (see proposal). More specifically, I decided to use a dataset of classical piano midi files and use them to train some deep learning setup that I could use for music generation.

In the previous milestone report 1 (see here), exploratory data analysis was presented. Furthermore a framework to generate music was presented. We note that a major takeaway from the previous report were the two helper functions we created: encode_song() and decode_song(). The encode_song function took a midi file and represented it as a list of note events and goforwards events (see first Milestone report for more informaton). The decode_song function took an encoded representation of a song and returned a midi file.

In this report, we look at a simplification of the generation scheme presented in Milestone 1 and assess results. All code pertinent to this report can be found in the Milestone2 jupyter notebook.

Music Generation Scheme:

Every midi file can be represented (encoded) as a stream (list) of note and goforwards events. For example below we see the representation of the first few events of Beethoven's Fur Elise.

['76', 'gf0.25', '75', 'gf0.25', '76', 'gf0.25', '75', 'gf0.25', '76', 'gf0.25', '71', 'gf0.25', '74', 'gf0.25', '72', 'gf0.25', '45', '69', gf0.25', '52', ...]

The above somewhat resembles a time series in that the each entry is an event that precedes the next event (so the list is ordered in time). That is not exactly true however since we could in fact reverse the order of the 45 and 69 in the second line of the encoding shown above and have the same representation. More specifically, the order of note events between gf (goforwards events) does not change the meaning of the representation. This can cause learning to be difficult since the representation itself is not unique. A way to overcome this is to represent each combination of note events (say 45 and 69) that occur between gf events as a note event itself. This however would cause the vocabulary of the representation to be vast (think how many combination of notes 10 fingers can play on a piano).

In this report we keep to the encoding discussed in Milestone 1 while listing the note events at a moment in time (i.e. between goforwards events) in increasing order of pitch (thus adding some form of structure to the representation). Furthermore, we train a recurrent neural network to predict the 51st event given a sequence of 50 events. The recurrent neural network can thus predict the 51st event as either a note event (represented by a pitch number), or a goforwads event (which can be something like gf0.25).

The drawback in this approach is that since the majority of the events in a stream are goforwards events, the network might be biased to predicting these. A scheme presented in the first milestone report may be able to tackle this problem (albeit a little bit more involved for training), however we keep that for the final report.

Preparation of Inputs and Outputs:

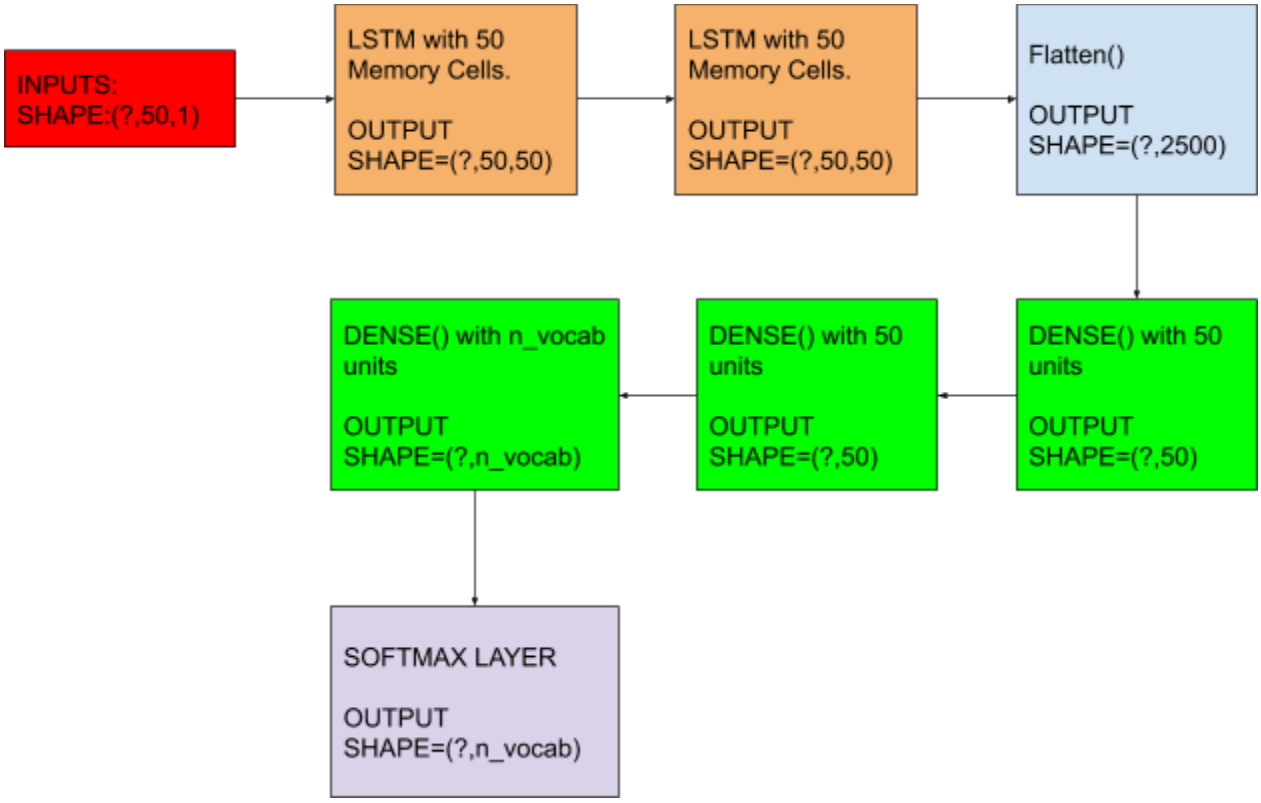
Previously (in the prepare.ipynb jupyter notebook) we encoded all the songs in our database and ordered them in a dataframe (that can be found in encodings/information.pickle). In order to train a neural network, the stream should be transformed into a stream of numbers rather than a stream of strings (integer encoding). We do this by first obtaining the set of unique events occuring in all the representations (i.e. obtaining the vocabulary), and then defining transformations from strings to integers and vice versa (see the EventToNumber and NumberToEvent dictionaries in the jupyter notebook).

Now having the appropriate inputs for a neural network we prepare the sequences of inputs and their corresponding output. Taking a close look, every input sequence will have 50 integer elements, and one event as an output. For the output, we opt to one-hot-encode the output rather than integer encode. This would entail having an output layer for the network that has as many nodes as words in the vocabulary (with softmax activation).

The preparation of the inputs and outputs can be found in the function prepare_inputs which takes as input the data frame including the encoding of the songs, the number of words in the vocabulary, and the EventToNumber transformation.

Neural Network Architecture:

In this report, we investigate the use of a LSTM neural network. Specifically, we opt for the following architecture:



Training the Network and Results:

To speed the process of training we use CuDNNLSTM instead of LSTM. CuDNNLSTM is a fast LSTM implementation using CuDNN. CuDNNLSTM requires GPUs however, consequently the notebook was run on an aws p2.xlarge server.

To simplify training and analysis of results, we limit the training data to one artist, namely Mendelssohn. Mendelssohn has 15 songs in the set we have. This translates to 29240 input sequences (not so small of a dataset). We take 14 of these songs for training, and keep one to see how the model performs in guessing the next note for the unseen song. We note the test train split is small here, however with the intention of music generation that should be fine.

The neural network is trained for 500 epochs (with a batch size of 1000). The training takes around 12.5 minutes achieving a loss of 0.0788. The accuracy score over the training data is about 98%.

Looking at the unseen song, the model performs with an accuracy of 16.25 percent. To make sure this result is not due to the model simply guessing one very likely event many times, we plot the distribution of predicted events for the testing data (see figure 1). As can be seen the distribution seems to be reasonable. Events like gf0.5 are very likely and happen the most. Note events are also predicted relatively frequently with the distribution being pretty uniform over the notes.

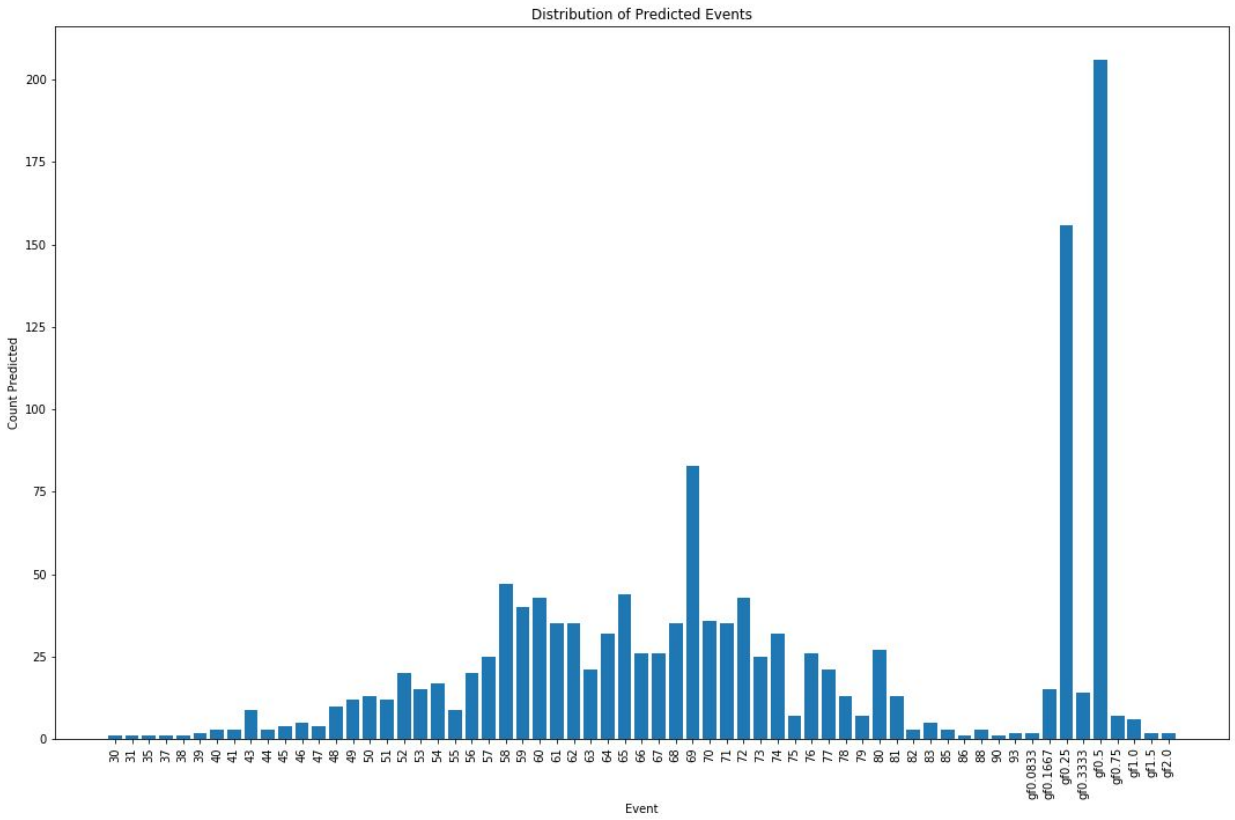


Figure 1: Distribution of Predicted Events for Unseen Input Sequence

Generating Music with the Trained Network:

Having a trained network, we now:

1. Feed the network a sequence of 50 events.
2. Let the network predict the 51st event.
3. Repeat with the last 50 events in the sequence until we obtain a sequence of desired length.

The function generate_music does the above.

We generate many samples using input sequences that the model saw during training (these sample can be found in the Samples folder with the name Mendehlsson_TrainingDataSampleX.mid) as well as with input sequences the model never saw during training (these sample can be found with the name Mendehlsson_TestingDataSampleX.mid)

Conclusions:

The generated songs actually do show some form of structure. Those generated from input seen in training are actually quite pleasant, and differ a little from the original music (but do not differ much). Those generated from unseen input tend to start rough but get better as the sequence progresses (possibly becoming more similar to something in the training set).

For the next report, quantifying the generations based on how similar (the less similar the better) they are to songs from the training set is considered.