DJamBot: Music Generation with Music Theory and Dynamics

Daniel Dore
ddore@stanford.edu

Joey Zou
zou91@stanford.edu

Abstract

We studied the problem of using deep learning to compose new music. Our model produces polyphonic music with dynamics, mimicking human classical piano performances. We use two separate neural networks to first generate a chord progression, then generate notes to fit this progression.

1 Introduction

Music is a fascinating domain for innovation: it combines a deeply intuitive, creative process with a large amount of mathematical theory and sophisticated technology. The possible domains of application are endless and diverse, including such disparate domains as music classification, synthesis of new sounds for electronic musicians, automatic transcription, and many more. We chose to focus on the problem of AI-powered music composition. In popular culture, music composition is a creative task that we think of as deeply human: in a poignant moment of 2001: A Space Odyssey, HAL-9000 reflects some humanity by simply singing the children's song "Daisy, Daisy." It would have been ludicrous in the 1960's to dream of HAL-9000 writing his own swan song! AI music generation could also be a vehicle to make the task of the human composer easier. For example, a user could give the AI a melody, and it would return a 4-part harmony based on that melody.

Perhaps surprisingly, music generation turns out to be a task which is quite well-suited for AI. For centuries, the primary means of communicating Western music has been through the musical score. This document represents a piece of music as a sequence of precise notes, which can be thought of as a pitch combined with a duration, as well as additional information indicating articulation, dynamics, tempo, etc. This structure can be digitized in a variety of ways, turning a piece of music into some sort of time-indexed sequence of pitches. A very common digital music representation is the MIDI file. This is a standardized music representation which can be outputted directly from electronic musical instruments and can also be synthesized and manipulated in a variety of commercial software. MIDI represents a piece of music as a stream of note events: a note event can be the start or stop of a note, and include the pitch, duration, and velocity (i.e. the volume) of the note.

Our model consists of two distinct neural networks. The first neural network predicts the next chord in a chord progression, given the previous chords played. When generating music, this output is then fed back into the model as input, so that we generate an entire chord progression. The second neural network takes this chord progression, as well as the notes played previously, and predicts which notes should be played at a given timestep. More precisely, at each timestep, our model outputs a vector of probabilities, indicating which notes are to be played, and a vector of "velocities", indicating how loud each note should be. When training our model, we build a dataset in this form by pre-processing a set of MIDI files.

CS230: Deep Learning, Spring 2018, Stanford University, CA. (LateX template borrowed from NIPS 2017.)

2 Related work

- **JamBot**[1] generates polyphonic music by training two different models: one to learn chord progression, and another that learns how to generate polyphony given a sequence of chord progressions. Hence, the implementation involves training a chords model based on the training data's chords to create a model that can generate a sequence of chord progressions, before then training a polyphony model that learns the various voices of a piece based on the piece's chord progression sequence. The JamBot implementation has the advantages of being simple, able to accept MIDI files as training input, and able to produce polyphonic music; however, it does not consider dynamics and hence produces an output with uniform dynamics.
- **DeepJ**[2] generates polyphonic music with varying dynamics. Here, three outputs are learned simultaneously by one model: play probability (the probability a given note is played at a certain time), replay probability (the probability a played note is replayed immediately after it ends), and dynamics. The replay probability is a technical detail needed to specify a piece due to the usage of the "pianoroll" representation of data (see below for more details), so a simplified way to view the model is that it produces note-playing probabilities and corresponding velocities for dynamics. The model can also learn about different styles of composers, provided the training data is tagged with composer information.
- Google **Magenta**[3] is a comprehensive music generation project with many different models and implementations that take into account various musical aspects, each offering different advantages and limitations. Two particular implementations we found were the "improv_rnn" implementation, which generates melodic music given a choice of accompanying chords and hence can be seen as a simplified implementation of JamBot, and the "performance_rnn" implementation, which produces polyphonic music that incorporates dynamics.
- **DeepBach**[4] is a genre-specific music generator specialized in composing chorales in the style of Bach. One novelty in their approach is the use of a data representation that allows for the construction of entire harmonies at once, instead of generating the music sequentially.

In some sense, our project combines the implementations of JamBot and DeepJ to create an implementation similar to that of the "performance_rnn" implementation of Magenta without requiring the remaining framework of Magenta to run.

3 Dataset and Features

The popularity and versatility of the MIDI standard means that there are many online repositories of MIDI files available. We concentrated on two particular databases. First, there is the Lakh MIDI dataset [5], containing roughly 200,000 MIDI files representing a variety of genres, all scraped from the internet. One drawback of this dataset is that since the files are from many different sources, their quality is inconsistent: for example, some files might be lacking information on the velocity/volume of notes entirely (e.g. many files might be from amateur transcriptions of popular songs, rather than generated by an electronic instrument). Another drawback is that the files are from a number of different genres, so training a music generation neural network on this dataset might result in a stylistically agnostic performance. To address both of these drawbacks, we also looked at the Yamaha E-piano Competition dataset[6]. This consists of approximately 1400 MIDI files, which are recorded from expert performances of classical music on the piano. This means that they are all from the same broad genre, and that they all have velocity annotations.

From this MIDI dataset, we preprocessed the data to turn each file into a more usable format. Here we followed the approach of JamBot and used many of its preprocessing features, which includes functions that normalize the tempo to 120 beats per minute and key signature to C major/minor, as well as functions that compute histograms of notes played, which are useful for the chord progression model. The JamBot paper then uses tools from the pretty_midi library[7] that converts the MIDI files into what are called *pianoroll* arrays, which are two-dimensional binary arrays where the (n, t) entry denotes whether note n is played at time t or not (so its value is 0 if the note is not played, and 1 if it is played). Here the time can be discretized according to either eighth notes (4 time steps per second given a 120 BPM tempo) or sixteenth notes (8 steps per second); we experimented with both settings. This data representation is more amenable to feed in as inputs to a neural network than, say, the original sequential format of MIDI files consisting of events that have to be decoded first. (Notice this representation has the drawback that two repeated notes are indistinguishable from one held note, since both cases are modeled as a note being played for two time steps. The DeepJ implementation remedies this by introudcing a replay feature to distinguish between these two events; we have chosen to ignore this issue for this project.) We modify this representation to allow for dynamics by allowing the array values to take on any integer value between 0 and 127, representing the allowable velocity values in MIDI files. For the purposes of training, we normalized the velocity values by dividing by $max\_velocity = 127$, and multiplying back during generation.

After preprocessing, we ended up with 360 usable modified pianoroll files (some files were unable to be processed since they did not have a clear tempo or key signature, in which case the preprocessing program ignored it when normalizing the tempo and key signatures). We split the files into 90% (or 324) for training and 10% (or 36) for testing.

4 Methods

We followed the approach of JamBot, which used *recurrent neural networks* (RNN) comprising of *long short-term memory* units (LSTM for short). The specific implementation used two neural networks, one to predict chord progression, and one to generate notes based on a generated chord progression. Both of those networks used one LSTM (long layer followed by one dense (i.e. fully connected) layer and one sigmoid activation layer, and the loss computed as a categorical cross-entropy loss. Since we wanted to generate dynamics, which affects note generation but not chord progression generation, we decided to keep the first network from JamBot as is and modify the second network.

Since we want to predict both the notes played and the velocities of the played notes, we used a custom loss function. We treated learning which notes to play as a series of binary classification problems: is note n on or off? Thus, we used binary cross-entropy as the loss function for the probability vector. For the velocities, this loss function is inappropriate, so we used mean-squared error. However, using just mean-squared error results in poor performance, as it over-penalizes the model for predicting non-zero velocities for notes which are "off". Thus, following the approach in the DeepJ paper, we only include terms in the mean-squared error computation corresponding to notes which are on:

$$\mathcal{L}(n_{\text{true}}, n_{\text{pred}}, v_{\text{true}}, v_{\text{pred}}) = \frac{1}{N}\left[\sum_{i=1}^{N}(n_{\text{true}}^{(i)}\log(n_{\text{pred}}^{(i)}) + (1 - n_{\text{true}}^{(i)}\log(1 - n_{\text{pred}}^{(i)}))) + \sum_{i=1}^{N} n_{\text{true}}^{(i)}(v_{\text{true}}^{(i)} - v_{\text{pred}}^{(i)})^2\right].$$

In addition to changing the loss function, we experimented with changing the architecture of the note generation network, including using 2 LSTM layers instead of 1, adding dropout between the LSTM/dense layers, and making the LSTM(s) bidirectional.

5 Experiments/Results/Discussion

Given the very human nature most would associate to music, the ideal method to evaluate a model for music generation is to conduct a "Turing test" where participants listen to both an actual song/performance and a model-generated output and decide which piece sounds more human-like or desirable. However, despite experimenting with a variety of architectures, input representations/normalizations, and parameters, we have been unable to produce a model with all of our desired features that produces meaningful music. Several of our models have produced very low probabilities of playing any notes whatsoever, very low velocities, or music that sounds like a single note being played.

In lieu of results we can evaluate, we instead look at the losses produced by the model as a replacement metric for our models. Listed below are the results from three of the models we tested: one model with the same architecture as JamBot (one LSTM plus one dense), and two models with two LSTM

Model	Epochs	Learning rate	Train error	Test error
1 LSTM	25	1×10^{-5}	0.0323	0.0362
2 LSTM	20	5×10^{-6}	0.0394	0.0408
2 LSTM (+ bidir.)	25	5×10^{-6}	0.0361	0.0374

Table 1: Table of models used

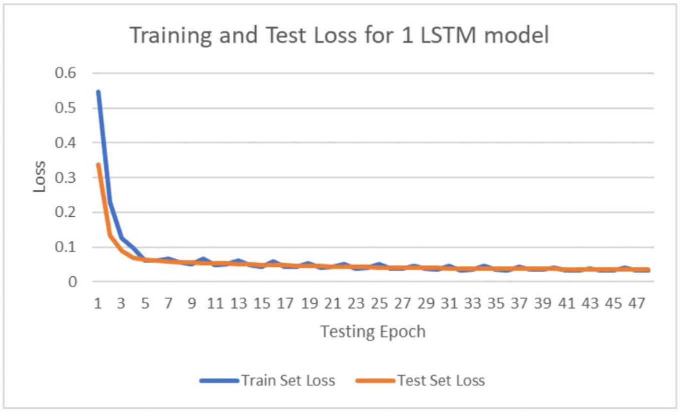

Figure 1: Plot of training and test losses for the model with 1 LSTM layer

layers and 80% dropout layers following each LSTM layer, of which one model had bidirectional LSTM layers and the other did not. While the output was not desirable, the models themselves seemed to work as the loss functions converged nicely. Each model used 324 training files and 36 test files, and each model used the Adam optimizer. Notice that in the first two models testing occurred twice per epoch, while in the third model testing occurred once per epoch. On the AWS servers, each model took between 15-20 minutes to train per epoch.

6 Conclusion/Future Work

In the end, we were unable to get our model to produce output that realistically sounded like music. While we got nicely converging losses, the output wasn't desirable: either very few notes were played because either the probabilities or the velocities were predicted to be very low, or the music produced did not deviate much from a constant note. We also tried using other neural network architectures, such as using 2 LSTM layers and adding Dropouts in between, or changing the LSTMs

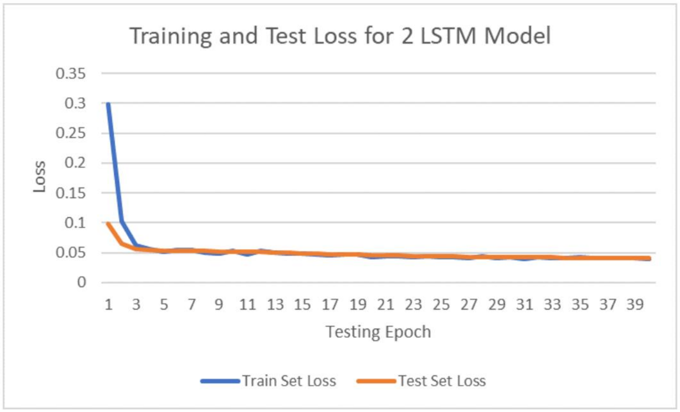

Figure 2: Plot of training and test losses for the model with 2 LSTM layers (unidirectional)

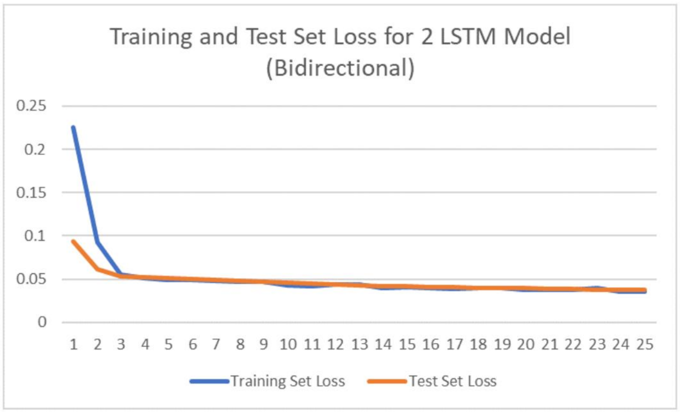

Figure 3: Plot of training and test losses for the model with 2 LSTM layers (bidirectional)

to be bi-directional, and the same things happened: while we got good losses, the actual outputs were not good.

We suspect that there may just be an issue with the data processing part rather than the neural network itself, since our models did train well, and the various versions of the architecture did not change the output very much. Given more time, we would like to debug this a bit more and hopefully produce music played with appropriate human-like dynamics.

One possible explanation for the poor performance is that the data we synthesized from the MIDI files was sampled incorrectly, leading to input data where a majority of notes are silent. In order to produce input for the chord-training model, the data-processing in the original JamBot implementation manipulates the original MIDI files by "normalizing" them. This means that it first attempts to determine the key and tempo of the input recording, and then normalize the tempo to a fixed value and transpose the key to C. Then it attempts to assign a chord to each measure of music, which is used as part of the input to the second LSTM. Our dataset primarily consisted of virtuosic piano performances, so it is possible that the data processing classes were less able to determine a single correct key/tempo or a single correct chord for each measure. If the tempo was incorrectly determined to be much too fast, it is possible that we would have primarily sampled silence (i.e. a majority of time steps would have no notes). However, the original JamBot model did train correctly on this dataset, so this is not the entire problem.

For future work after resolving the current issues, we would like to consider adding an attention mechanism to the LSTMs, to allow for more long-term structure considerations by the model.

7 Contributions

The two authors wrote separate sections of the final report and poster. Dan did most of the research into existing implementations, and both authors contributed to the ideas behind this implementation. Joey produced the majority of the plots and tested and evaluated most of the models, as well as delivering the poster pitch. Dan and Joey wrote different parts of the code at different times.

Our code can be found on GitHub, at `https://github.com/dorebell/DJamBot`.

References

[1] Gino Brunner, Yuyi Wang, Roger Wattenhofer and Jonas Wiesendanger. “JamBot: Music Theory Aware Chord Based Generation of Polyphonic Music with LSTMs.” `https://github.com/brunnergino/JamBot`

[2] Huanru Henry Mao, Taylor Shin and Garrison W. Cottrell. “DeepJ: Style-Specific Music Generation.” `https://github.com/calclavia/DeepJ`

[3] Magenta: Music and Art Generation with Machine Intelligence. `https://magenta.tensorflow.org/`, `https://github.com/tensorflow/magenta`

[4] Gaëtan Hadjeres, François Pachet, Frank Nielsen. "DeepBach: a Steerable Model for Bach chorales generation". ICML 2017.`https://github.com/Ghadjeres/DeepBach`

[5] Colin Raffel. "Learning-Based Methods for Comparing Sequences, with Applications to Audio-to-MIDI Alignment and Matching". PhD Thesis, 2016. `http://colinraffel.com/projects/lmd/`

[6] Yamaha Piano E-Competition, `http://www.piano-e-competition.com/`

[7] Colin Raffel and Daniel P. W. Ellis. Intuitive Analysis, Creation and Manipulation of MIDI Data with pretty_midi. In Proceedings of the 15th International Conference on Music Information Retrieval Late Breaking and Demo Papers, 2014. `https://github.com/craffel/pretty-midi`

[8] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Rafal Jozefowicz, Yangqing Jia, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Mike Schuster, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[9] Chollet, François et al.. Keras. 2015. `https://keras.io`