Music Composition with RNN

Jason Wang
Department of Statistics
Stanford University
zwang01@stanford.edu

Abstract

Music composition is an interesting problem that tests the creativity capacities of artificial intelligence. Creating original pieces of music is not much different than generating free text or any other form of sequential data such as stock price trends. We apply simple algorithms such as the n-gram model to explore the space of music composition. Then we explore the ability of the RNN and the LSTM in generating original and creative pieces of music.

1 Introduction

In this study, we look at several different approaches to teach a computer to generate music in the style of Irish folk music. Such algorithm would be useful for composers looking for inspiration to fix writers' block or for enthusiasts who want to mimic the quintessential style of a particular genre of music. Generating music from midi file input is a problem that captures the challenges of working with temporal data. Recent advances with Recurrent Neural Nets in the field of classifying sentiment of text, predicting trends in financial time series, and generating text motivate us to apply such approaches to music. We feed short fixed length of segments representing sequences of notes to a many-to-one RNN in hopes to classify the next note played after the sequence. In doing so, we hope the RNN will learn dependencies between notes and the conditional probability of notes in sequence so that we can generate new and original sequences.

1.1 Problem Definition

Given a fixed time series $x_1, x_2, ..., x_T$ of features where x_i represents the note's pitch played at the ith time-step, predict the pitch played at x_{T+1}.

To generate music, we give initialize the seed as $x_1, x_2, ..., x_T$ and evaluate the RNN to predict x_{T+1}. Then iteratively, feed the most recently generated T notes to to the RNN to predict the subsequent note.

2 Related Work

One of the first attempts to compose music used a single-step prediction. Essentially, the algorithm predicts the note at the $t + 1$ time-step given the note at the t time-step as inputs. After learning has converged, the network can be seeded with initial input values–written by human–and iteratively generate notes one by one, using the newly generated notes as subsequent inputs. These approaches were first pioneered by Todd (1989) and Stevens and Wiles (1994), and Mozer (1994).

Perhaps the simplest variant of note-by-note generation algorithms is the bi-gram model where notes are generated stochastically based on estimates of the probability of the x_T, note at time-step T,

given x_{T-1}. Such can be estimated by a MLE-like count of the number of times the pair (x_{T-1}, x_T) is observed in all of the training data divided by the number of times x_{T-1} occurs with Laplacian smoothing if necessary.

RNN-LSTM are widely used in the area of text classification and generation. RNN's with over 90% accuracy at classifying sentiment have been trained, converging in a few epochs and thus requiring minimal training (Brownlee, 2016). Random text generation has led to humorous attempts and could also be used to attribute the author of texts with better success rates than SVM, HMM, and other standard techniques that don't take advantage of sequential data. Likewise, RNN has also been applied to model polyphonic music (Boulanger-Lewandowski, 2012). It has also been used to study the relationship of chords and melody sequentially (Eck, 2010). Drawing inspirations from these works, we apply several RNN-like algorithms to model sequences of musical pitches.

3 Data

We trained our algorithms on the entire Nottingham files of 912 songs, each over a minute long. We use a 70-10-20 split between training, validation, and test set. While test set is not necessary at all since our objective isn't to reproduce music exactly, the error comparisons give us a metric to benchmark our algorithms.

To process the data, we extract the melody from each song and divide each measure into eight time-steps. Then we transposed each melody to C major (or a minor). Since the different key signatures are merely translations of music to different pitches, transposing retains all musical qualities while reducing the number of commonly observed notes. For each time-step i, we add the sequence $v_1, v_2, ..., v_{i+T-1}$ to the dataset of training features with the assigned class v_{i+T} if such notes exists. We choose $T = 32$ so that our sequences are exactly four measures. This gives us a total of 154992 training sequences of length T.

4 Features

The only explicit feature is pitch quality of the note at a given time-step. Pitch quality is represented by a integer ranging from 21 to 109 inclusive. Each half step is represented by a increase in pitch by one. For instance middle C is 60. The $C\#$ above middle C is 61. The range 21 to 109 covers all the keys on the piano from the lowest A to the highest C. The pitch quality of a rest (no notes played) is arbitrary assigned to 0. It doesn't matter what value we assign since we don't feed the sequence of pitch values into the RNN. Instead we map each pitch to a randomized integer key from 0 to the number of unique pitches and normalize so that the values are between 0 and 1 inclusively. Randomizing ensures there is no correlation between neighboring pitches, which decreases training error. Normalizing optimizes training where we use a sigmoid activation function.

Implicitly, we encode features such as the duration of the notes by the number of consecutive times the same note is repeated. For instance, the sequence C, C, C, C, G indicates that C is held for 4 time-steps whereas G is only played for one time step. Sequences also encode transition probability of notes. For instance, the sequence C, E, G is much more likely to appear in music than $C, F\#, D$. Ideally, the RNN can learn which transitions are favored over others.

5 Methods

5.1 N-gram

The N-gram model is a simple music generator. Although it deviates from our previous discussion of RNN in that it doesn't update any internal parameters to improve its prediction of subsequence notes, it serves not only as a benchmark to evaluate our other algorithms but also as a simple algorithm to generate new music.

From the training set, we examine all sequences of notes for a given number of time-steps n. This gives us a massive dictionary of all short sequences of musical expressions and the probability each phrase is used in music. Let v_t denote the note played at time-step t. Then we can estimate the probability of the next note by the following

$$p(v_t|v_{t-1}, ...v_{t-n}) = \frac{[v_{t_n}, ..., v_t] + \lambda}{[v_{t_n}, ..., v_{t-1}] + k\lambda}$$

where λ is the smoothing constant and k is the total possible number of values x_t takes. The notation $[x_{t_n}, ..., x_t]$ denotes the number of times the particular sequence of notes was observed in the training data.

5.2 RNN

We use a many-to-one RNN, with inputs of fixed length sequences and train a classifier to correctly classify the note at the subsequent time-step into one of 88 possible values, each corresponding to a valid piano note. Many-to-one RNN read input sequences from start to end one at a time, updating the hidden values in a feedforward fashion. By the end of the sequence, it predicts the output, compares it to the actual output, and we backpropagate to update the parameters accordingly.

The parameters we train are W_{hh}, W_{xh}, W_{hy} where each is a matrix and bias vectors b_h and b_y. We also have the hyperparameter T to denote the fixed length of input sequences.

The input layer is x_i for $i = 1, ..., T$. The hidden layers are $h_i \in [-1, 1]$ for $i = 0, 1, ..., T$. We perform the following update for each $i = 1, ..., T$, initializing $h_0 = 0$.

$$h_i = tanh(b_h + W_{hh}h_{t-1} + W_xhx_i)$$

Essentially, at each i, we compute the next hidden state by taking a linear combination of the previous hidden state and the current input state. Then we apply an activation function to smooth it so backpropagation is easier.

Finally, we compute the output

$$y = b_y + W_{hy}h_T$$

5.3 RNN-LSTM

The RNN-LSTM preforms the exact same process as the RNN except with a more complicated update step. It includes numerous gating functions and additional parameters. The update step is as follows.

$$f_t = \sigma(b_f + W_{hf}h_{t-1} + W_{xf}x_t)$$

$$i_t = \sigma(b_i + W_{hi}h_{t-1} + W_{xi}x_t)$$

$$o_t = \sigma(b_o + W_{ho}h_{t-1} + W_{xo}x_t)$$

$$c_t = f_t \circ c_{t-1} + i_t \circ \sigma(b_c + W_{hc}h_{t-1} + W_{xc}x_{t-1})$$

$$h_t = o_t \circ \sigma(c_t)$$

Where W, b are parameter matrices and vectors. x_t is the input. h_t is the hidden state. c_t is the cell state. f_t is the forget gate vector or the weight of remembering old information. i_t is the input gate vector or weight of acquiring new information and o_t is the output gate vector.

Table 1: Evaluation of algorithms for music generation

Model	Log-loss	Training Accuracy	Test Accuracy
Random	–	0.0243	0.0244
3-gram	–	0.3150	0.2951
RNN (50 epochs)	1.0169	0.6269	0.4822
RNN-LSTM (50 epochs)	**0.7460**	**0.7332**	**0.5888**
RNN-LSTM with 2 layers (6 epochs)	1.4889	0.4941	0.4019

6 Results

6.1 Experiments

We train the RNN where the hidden layer is a 64-dimension vector. Recall that we are classifying subsequent notes in one of 88 classes so we apply the softmax function to output the most likely class. Since the problem of generating music is essentially the problem of note classification, we seek to minimize the multiclass log loss (cross entropy) and use ADAM optimization algorithm for speed. While we are not interested in classification accuracy, we will still record it to benchmark our various algorithms. Training an algorithm with perfect training and testing accuracy is not our goal because the machine will instead memorize music sequences instead of generating music creatively and as a result, we leave some margin for error. In the RNN-LSTM, we add a dropout of 0.2.

We train each RNN on 50 epochs with batch size of 128. Initially, we train various RNN for 10 epochs to determine which hyperparameters minimize the training loss by the end of 10 epochs. With grid search, we determined that fixing the length of input sequences to 32 dimensions and hidden values to 64 dimensions yields the best results. After each batch, we backpropagate to minimize the loss function defined as

$$-\frac{1}{N}\sum_{i=1}^{N}\sum_{i=1}^{M} y_{ij}\log p_{ij}$$

where M is the number of labels, N is the size of training set, y_{ij} is binary indicator of instance i is labelled correctly, and p_{ij} is the model probability of assigning label j to instance i.

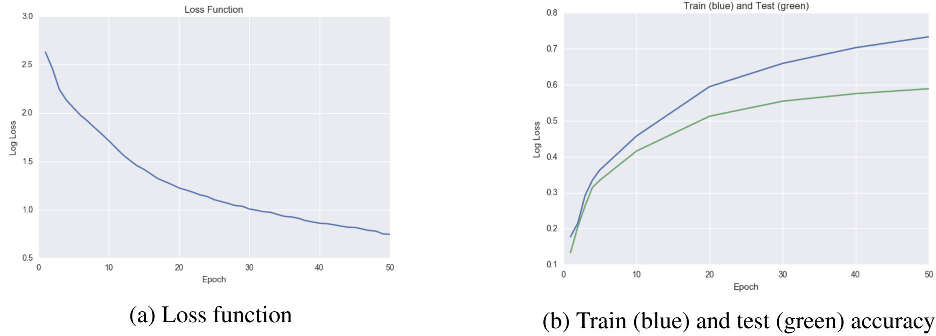
We compare all algorithms to the baseline, which is the random generation of notes. We have the n-gram models, the RNN, the RNN-LSTM, and the RNN-LSTM with 2 layers of LSTM (second layer is the exact same as the first layer). Also note, the 3-gram was selected because it performed best on the validation set. See Table 1 for results.

6.2 Discussion

LSTM learned the task of melody generation really well. Each epoch took 450 seconds for the LSTM on a 2.6 GHz Intel Core i7. RNN was about 2 times faster and the LSTM with 2 layers was about 3 times slower. The RNN-LSTM loss function decreased quickly at first but continued to decrease throughout all of training. In fact, training and test error both decreased throughout training so it's very likely that we would achieve better results had we ran the LSTM for more epochs. The RNN exhibited similar patterns. The LSTM with 2 layers achieved the minimum loss at 6 epochs, then failed to converge afterwards.

To generate music, we feed a segment of music of 32 time-steps rom a randomly selected piece in the test set to ensure the LSTM isn't memorizing music sequences we've trained it on. We then feed forward update to generate the subsequence note. The most recent sequence of 32 time-steps is then feed iteratively into the LSTM and the process can continue indefinitely.

Even the simple 3-gram model was able to produce pleasant sounding music. This reveals that Nottingham melodies have a highly Markov-like structure. However, the 3-gram has no sense of



(a) Loss function

(b) Train (blue) and test (green) accuracy

Figure 1: RNN-LSTM Learning

longterm dependencies. The LSTM was rather successful at learning longterm dependencies, the most remarkable of which being repetition of musical ideas in a structured manner. Some others are described below.

1. Chord progression: In all samples, the music learns the correct cadences such as the ubiquitous I-IV-V-I progression. In fact even if we seed a piece that begins on the dominant chord (V), the machine learns to resolve to the tonic (I).
2. Melody: The melodies are very lyrical and very much similar in style to Irish folk music. There are lots of short steps and alternating between ascending and descending. Wide jumps and dissonant notes, which were evident in the early phases of training, disappeared after more epochs of training.
3. Repetition: This is the true testament to the success of LSTM. Nottingham music is very repetitive and in almost all pieces, the melody is phrased in an "question-answer" fashion. An initial phrase would comprise of the first 4 or 8 measures and a very similar phrase would resolve the phrase to its tonic. The LSTM was great at generating creative ways to resolve previous melodies. In the example below (Figure II), the phrase from 6 to 12 seconds mirrors the introductory phrase. After the 12 second mark, it generates its own melodies.

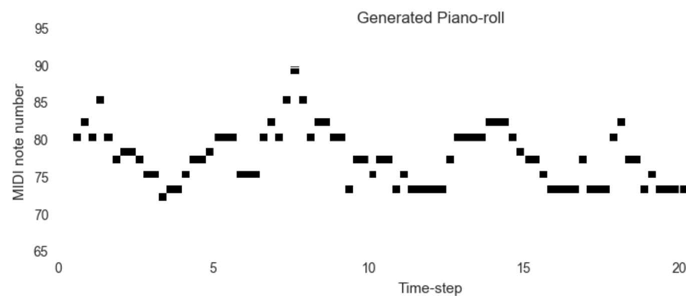


Figure II: Generated Music Sample

7 Future Work

Music composed by humans is often really structured and features reiterations of music ideas. LSTM recurrent neural nets is great for capturing long term temporary dependencies while allowing creativity in the short term. We expect classical machine learning algorithms such as SVM, random forests, and regressions to perform worse since they assume independence of temporally correlated features. While we had success with simple melodies, it would be great to apply the similar approaches to more complex music and run for more epochs until convergence. We can also investigate ways to encode rhythm. In addition, we can explore ways to generate harmony given melody and hope that machines can learn complex musical ideas such as counterpoint. This will require creative loss functions but similar infrastructure.

References

[1] N Boulanger-Lewandowski, Y. Bengio, P. Vincent, Modeling Temporal Dependencies in High-Dimensional Sequences: Applications to Polyphonic Music Generation and Transcription, in Proceedings of the 29th International Conference on Machine Learning (ICML), 2012

[2] K Goel, R Vohra, and J.K. Sahoo, Learning Temporal Dependencies in Data Using a DBN-BLSTM

[3] J, Brownlee, "Time Series Prediction with LSTM Recurrent Neural Networks in Python with Keras". http://machinelearningmastery.com. June.2016./ [Online; accessed 10-November-2016].

[4] C.Olah, "Understanding lstm networks." 'http://colah.github.io/posts/2015-08-Understanding-LSTMs/, Aug.2015. [Online; accessed 10-November-2016]

[5] Eck, D. and Schmidhuber, J. Finding temporal structure in music: Blues improvisation with LSTM recurrent networks. In NNSP, pp. 747756, 2002.

[6] Zhang, X and Lapata, M. Chinese Poetry Generation with Recurrent Neural Networks. EMNLP. 2014.