CS230

# LSTiestoM: Generating Classical Music

**Daniel Gallegos**
dgalort@stanford.edu

**Sean Metzger**
seanmetz@stanford.edu

## Abstract

We present a method for generating polyphonic MIDI sequences using Deep LSTM networks.

## 1 Introduction

Composing music is one of humanity's most amazing accomplishments. We thus sought to generate high quality classical music using an RNN. As musicians, we were intrigued by the possibility of the creating new songs using deep learning. Although many LSTM networks can generate decent music, We sought to create one that could not only support multiple notes at once, but also encode dynamics, making the music sound less robotic and more human generated.

We found a large data set of classical music in midi format. We encode 100 note sequences in by note index and have our RNN output the next note based on the last 100. This is a character level RNN problem, much like the Dinosaur example on Coursera [11]. Due to this, we chose an LSTM-based architecture over a vanilla RNN because it is able to remember information stored about previous inputs. We also drew a lot of inspiration for how to create our model from the Jazz improvisation lecture [12].

We trained our model by trying to predict the next note in the training set based off the last 100. Then to generate the final output, we run the neural net on a random sample from our training set and let it predict the next note at every time interval and append it to the input, running our network repeatedly until it has generated an entirely new MIDI sequence. We output this sequence to a MIDI file which encodes a new song, which can then be played using a piano voicing.

## 2 Related work

RNNs are the current state of the art in music generation, as the learning of time dependencies makes them ideally suited to this task relative to statistically based music generation models, and they can be beautifully implemented using deep learning [8] [14].

### 2.1 LSTM based approaches

Google's Magenta team has generated very good music employing LSTMs. What distinguishes their model from the rest is that it incorporates dynamics, which allows them to generate the most human sounding music we were able to find [13]. They add a one-hot vector including dynamics to the end of the vector they fed into their RNN, like in our first two encodings. They've also combined their networks output with reinforcement learning techniques to reward creating music in specific genres, although that was outside the scope of this course [5]. A common problem with LSTMs is that they are great at generating short sequences of notes, but cannot easily encode long structure, which Google has attacked by inputing one hot vectors that contain information from the current time

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

step and from one measure ago in a LookBack RNN [1]. All the approaches we found were based on RNNs and LSTMs, so we chose to focus our efforts there, and the best work was being done on MIDI files rather than audio data, since MIDI richly encodes audio information in an easy to learn format. These approaches gave us tantalizing ideas for model improvements and a basic framework to build ours upon.

### 2.2 GRU approaches

We also found that GRUs are commonly used in music generation, and have very good performance, sometimes beating LSTMs on music generation tasks, which inspired us to use them as a potential architecture for our final project [7].

## 3 Dataset and Features

Our dataset consisted of midi files of classical piano songs by composers like Chopin, Beethoven, and Mozart, which were recorded by young pianists for a Yamaha E-Piano competition. These human generated midi files have human note timings, dynamics, and style, making this dataset ideal for generating natural sounding music relative to other MIDI datasets. [10]. We obtained $338$ songs of varying length in midi format from here totaling over $24$ hours of playing time. We created a development set and test set from this data by selecting a group of songs to be a development set and a separate group of songs to be the test set. We always tried to keep the test set to be roughly $5\%$ of the development set.

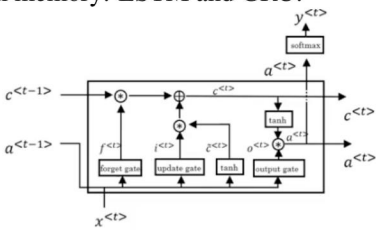
The midi format is a widely used format to efficiently store any type of electronic music. There are many different pieces of software available, such as Ableton Live or `onlinesequencer.net`, that can be used to listen and edit these files. The midi format consists of a list of messages that signal note events occurring in the song. All of these messages have a *time* field in it where it gives the relative offset of the current message with respect to the previous. This way every message has a relative time stamp. The most important message type, at least for our project, is the *note_on* type. This message has two important fields which are *note*, which specifies which key on the piano it is referring to $(0 - 127)$, and *velocity*, which specifies the dynamics of the note $(0 - 127)$. One can also have multiple tracks that contain these message lists in the midi files to encode multiple instruments, but we didn't use this functionality in our project.

We created a pipeline to transform our data to the specific encoding we were using and back to midi format, so we could play the output. We did this using the mido and music21 libraries in Python [6][9].

## 4 Methods

Since we decided to do a character level music generation algorithm, using a plain RNN probably would not yield good results. This is because RNN's can't store much information to pass it to the next cell. In practice this causes the RNN to only use the information from the previous state and the input to predict the output. However, since we know that musical notes and melodies are more complex and rely on large sequences of notes - with a 'grammatical' structure not unlike the English language, modeling it with a plain RNN was out of the question. There are two related methods we considered in order to give our RNN a longer term memory: LSTM and GRU.

$$\tilde{c}^{<t>} = \tanh(W_c[a^{<t-1>}, x^{<t>}] + b_c)$$
$$\Gamma_u = \sigma(W_u[\, a^{<t-1>}, x^{<t>}] + b_u)$$
$$\Gamma_f = \sigma(W_f[\, a^{<t-1>}, x^{<t>}] + b_f)$$
$$\Gamma_o = \sigma(W_o[\, a^{<t-1>}, x^{<t>}] + b_o)$$
$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + \Gamma_f * c^{<t-1>}$$
$$a^{<t>} = \Gamma_o * \tanh c^{<t>}$$




LSTMs are ideal for this task because they can learn patterns of long term dependencies in data [12]. Like in speech, musical ideas can depend on notes that were input many time frames ago. As you can see in the figure above, LSTMs receive the current input (x), memory cell input from one time

step before (c), and the activation from one time step before (a). In short, an LSTM learns forget, update, and output gates that help it use these inputs and pass inputs along to the next cells to learn time dependencies in the data. A GRU cell is much like an LSTM cell, just with a merged hidden and cell state, and has a simple 'update gate' instead of forget and input gates.

In order to code up these two different methods, we first began by using plain TensorFlow RNN Cells and used that to train models for the first two encodings [4] [3]. However, we found that for quickly iterating and trying new things TensorFlow tended to be cumbersome and so for our last and final encoding we decided to use Keras instead since it allowed for very easy use, hidden layer expansion, and layer stacking, especially when adding regularization [2]. We also used either the *Adam* or the *RMSprop* optimizers to train our model, with the latter having quicker, better results. Keras' CuDNNLSTM and CuDNNGRU cells proved critical for quick iteration.

## 5 Experiments and Results

In total, we ended up training dozens of different models with different parameters and algorithms. Since a lot of the major changes we did to the models were in the encoding, comparing the loss function between encodings wasn't a feasible since we couldn't directly compare them. This is why despite using cross entropy loss to train our models, accuracy was a good evaluation metric. We would also subjectively listen to the output of our model because especially at the beginning our test metric values were all over the place and the only reliable metric was the generated song itself. When listening to the generated songs, we paid close attention to whether the notes seemed random, whether there was an underlying melody to them, whether they were in the same key, and whether they resembled the development set too closely. With our final model however, we were able to use the loss as an evaluation metric.

### 5.1 First Encoding

We thought the easiest way to do this was to mimic how midi files start and stop playing notes by having a *note_on* with positive *velocity* followed by one with $0$ *velocity*. So we had two one-hot vectors of size $128$ stacked on top of each other to symbolize all the notes that were being turned on and off at that given time step. To simplify our lives we chose a fixed time shift of $.01$ seconds like Magenta did [13]. We then trained this using an LSTM with $256$ hidden units. The results we got were abysmal. This is because the model would always output the same thing and cite really low probabilities of any note getting played. This made sense because the vast majority of frames were empty because of the small time step. Also, the small time shift at every frame made our model run incredibly slow and take up a lot of memory, so we ended up deciding to transition to the second model as we moved our encoding to more closely match that of the Magenta Project.

### 5.2 Second Encoding

Our second encoding consists of the same vector of length $256$ as in the first encoding, but with an additional $100$ row vector appended to the end of it. This additional vector encodes the time shift property, which is how Magenta encoded theirs. Each element in the represents the time between the current frame and the previous one in increasing increments of $.01$ seconds. This way we can encode a time shift of length between $.01$ to $1$ second. This method turned out to train a lot faster than the previous one. However, our results still weren't very good. We still found that the model didn't particularly care which element in the vector it output it made slightly larger as long as all of them were very small since there was still a very small probability of any element in the vector being $1$ at any given time. We decided we needed a radical change and after reading a blog post decided to completely pivot our encoding [15].

### 5.3 Third Encoding

We decided to simplify our encoding even further by encoding every time shift as being the exact same and just longer, giving every note an offset of 0.5. We also decided that instead of having two stacked vectors representing turning a note on and off, just having one vector that represents the sequence of notes that was played. Similar to the dinosaurus land coursera program, our models input was a 100x1 vector with the index of the last 100 notes/chords played. The output was a one

hot vector representing the next index played, which was used in our categorical cross entropy loss. We typically had a target one hot vector between 100 to 400 indices long depending on the size of dataset that we used. We knew this had the drawback that our model could never generate a group of notes that it had never seen before, but figured if our training set was large enough, we would have heard most of the common combinations that make songs. This turned out to be the best encoding we found and the one we began tuning hyperparameters for. Normalizing our input vector gave us a huge boost in accuracy on training and dev sets.

### 5.4 Tuning the Model and Model Results

We attempted to tune our RNN with several different methods. The first was our loss function. Since all of our vectors were boolean vectors, then we decided to use cross entropy as our loss function. We figured this was the most appropriate since it would take into consideration both how close our output at a certain index was to 0 and to 1 depending on whether it should be 0 or 1 respectively. We also tried using a weighted cross entropy where we set a different weight for the positive cross entropy. We believed that since our matrix was mostly 0's then it would be beneficial to upweight positive examples by the average number of positive examples in the data set. This gave us this as our loss where $\hat{y}$ is our model's output, $y$ is the real value, and $k$ is the positive weight chosen:

$$-\sum_{i=1}^{d}(ky^i log(\hat{y}^i) + (1-y^i)log(1-\hat{y}^i))$$

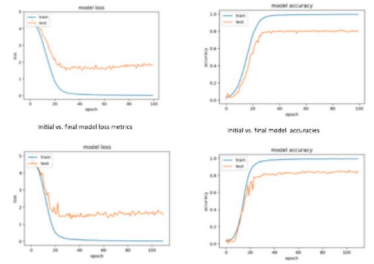
We found through our results that we were overfitting to the training data, so we decided to try out some regularization techniques. We tried adding dropout to our model and tried doing early stopping when we were training our model. We found out that early stopping didn't really affect the model very much, but dropout was a big help.

We also tried several levels of complexity of the model. We first created a LSTM with very few hidden units and then transitioned to testing more hidden units. Once we settled on 256 hidden units being the best, we decided to try out deeper networks, so we stacked two LSTM's and later tried four stacked LSTM's. This allowed our model to be able to learn more difficult things because we believed that at first our model wasn't doing well because it hadn't been able to properly learn how music worked. The details of our tuning and results can be seen below. We were able to greatly improve our model's performance through regularization and hyperparameter tuning!

| Initial Model Architecture Search | LSTM Cell (256), 1 Dense Layer | 2 Stacked LSTM Cells (256), 1 Dense Layer | 2 Stacked LSTM Cells (128), 1 Dense Layer | 2 Stacked LSTM Cells (512), 1 Dense Layer | GRU Cell (256), 1 Dense Layer |
|---|---|---|---|---|---|
| Train Accuracy | .994 | .995 | .995 | .995 | .9956 |
| Validation Accuracy | .771 | .8065 | .8065 | .8065 | .8141 |

| 'Panda' Model Improvement Highlights – | Dropout (p = 0.3) with 2 Stacked LSTM cells | Aggressive Dropout (p = 0.6) + Regularizing Kernel in LSTM | Stacked GRU Cells with Dropout | 4 Stacked LSTM Cells with 4 Dropout Layers, 1 Dense Layer |
|---|---|---|---|---|
| Training Accuracy | .995 | .986 | .986 | .995 |
| Validation Accuracy | .8241 | .8216 | .8216 | .8568 |
| Train Loss | .0402 | .0499 | .0499 | .0135 |
| Validation Loss | 1.56 | 1.52 | 1.52 | 1.518 |

| Hyperparameter choice | Sequence length | Learning Algorithm | Initial Learning Rate | miniBatch Size | Number of Epochs | Training Set Size |
|---|---|---|---|---|---|---|
| Final Value | 100 | RMS prop | .01 | 100 | 60 | 10,000 |




### 5.5 Manipulating Our Dataset

Another tuning parameter we tried, which wasn't necessarily exactly a parameter of the model was the distribution of the data set. At first we tried training and testing on random songs from our dataset.

We found this made it difficult for the model to generalize to new songs because the piano pieces ranged from the baroque to the romantic era. Furthermore, one can clearly tell that the style of Chopin is completely different than that of Mozart, so we decided that it would be beneficial to try to train on either just one Composer or if we can't to try to have the development and test sets come from at least the same era. This allowed our model to better focus on just learning one style of playing.

We attempted to tune our learning rate, by making it bigger so that our models would train fast, and a $\alpha = .01$ turned out to be the best. Both *Adam* and *RMSprop* anneal the learning rate, but our choice of alpha helped us learn quickly and also have good final accuracy. We also experimented with batch size, number of samples trained on and epochs.

### 5.6 Post-processing

The output from our models tended to not be perfect and needed some post-processing before they were ready to be listened to. Similar to many sampling algorithms, the output that our model generated needed some burn in time before it output interesting results. Therefore we typically got rid of the first $200$ frames that our model output since they were generally just the same note played in an infinite loop.

The output also tended to repeat the same note over and over again. This was because we selected notes based on the *argmax* of the probability that they should be selected. We hypothesized that this repetition of the same note was the model trying to put a rest into the sequence, but since it couldn't because of the way we set it all up, then we had to go in and manually remove them. This cleaning up of our final output resulted in a higher quality generated audio and is visualized in Figure **??**.

### 5.7 Generated Music

We found our models generated enjoyable piano music! Because we simplified our MIDI encoding to to get better model output and training accuracies, all our notes are the same length, but the music still sounded nice. You can listen to two of our best songs here: `https://soundcloud.com/seanaay/cs230`

## 6 Conclusion and Future Work

We found that our best model was the four stacked LSTM cells with dropout. This is based on the test accuracy we got back and an analysis of the generated songs that it produced. We think that some algorithms worked better than others because they allowed to encode the way that music works, which is fairly complex, without managing to overfit to the development set. We also found that the third encoding was the best one. This is because having to have the model select any combination of chords adds a lot of complexity and ends up not working very well. Also, having a fixed amount of time between predictions really helps the model not have to pick between a very large number of combinations of size of time shift and notes to play which is very hard for it to do.

There are many different directions we could go if we had more time and resources. We would add the functionality to have dynamics in our notes. This is something that is lacking and that we never tried. Without dynamics, our music ends up sounding robotic and not human. We could do this by adding to our encoding either a number or a one-hot vector to gauge how loudly to play each note. Another extension would be to add the ability to hold notes for longer. This could be done by making it that if a note is repeated then they are combined in the midi file for all of those intervals, which means that in this scenario we would have to actually encode pauses instead of relying on post-processing. We began this process in encoding 5. With these two extra things, our model could have the ability to produce any song out there because its output could be as expressive as any human could make it and more because it could also create music that is technically impossible to be played by a person. Other more minor future work that we could experiment with is to play more with the loss function we are trying to optimize. We could create a loss that takes harmonics into effect and prevents "bad" sounding combinations of notes, such as dissonance, by upweighting them. We could also expand our data set for each composer and train our model on each individual composer so that then we have generated models for each composer instead of a generated song for just classical music.

## 7 Contributions

Daniel did most of the work on the first two encodings and TensorFlow model, while Sean found the dataset, focused on the Keras work and the third encoding, carried out the hyper-parameter tuning, and setup AWS and GPUs. Sean did most of the poster and the writeup was split evenly.

## 8 Code

Here is a code as a zip file `https://drive.google.com/open?id=1NhO1IXx_5POu8QWvOlXwnUmsLgC2vJtb`

## References


[1] Generating long-term structure in songs and stories, Jul 2016.

[2] Keras. `https://keras.io/`, 2018.

[3] Numpy. `http://www.numpy.org/`, 2018.

[4] Tensorflow. `https://www.tensorflow.org/`, 2018.

[5] et. al. Adam Roberts, Jesse Engel. Interactive musical improvisation with magenta. *NIPS 2016*, Wed. Dec. 7 2016.

[6] Ole Martin Bjørndalen. Mido. `https://mido.readthedocs.io/`, 2017.

[7] Junyoung Chung and Caglar Gulcehre. Empirical evaluation of gated recurrent networks on sequence modeling. *arXiv*, 2014.

[8] Darrell Conklin. Music generation from statistical models. In *Proceedings of the AISB 2003 Symposium on Artificial Intelligence and Creativity in the Arts and Sciences*, pages 30–35. Citeseer, 2003.

[9] Michael Cuthbert. music21. `http://web.mit.edu/music21/`, 2018.

[10] Bernd Krueger. Classical piano midi page, 2016.

[11] Andrew Ng. Dinosaurus land, 2018.

[12] Andrew Ng. Jazz and improvisation, 2018.

[13] Magenta Project. Make music and art using machine learning, 2018.

[14] Sigur Skuli. How to generate music using a lstm neural network in keras, Dec 2017.

[15] Sigurður Skúli. How to generate music using a lstm neural network in keras, 2017.