Deep Learning Final Report: Deep Music Generation from a Single Song

Roland Huang, Will Merrill, and Jared Weinstein

April 2018

1 Problem definition

Deep learning has proven useful in a wide variety of applications, such as prediction systems, image recognition, and natural language processing. There has not been nearly as much research, however, in their application to the realm of art. Art is a uniquely human field. We decided to see if we could apply deep learning to music and attempt to create a recurrent neural network that can create pleasing melodies and harmonies.

This is a difficult task for a variety of reasons. First of all, the network must learn some representation of rhythmic structure and beat. Given only the times and pitches that notes are played, it can be a challenge for the network to learn to play notes at sensible times. The network must also learn some notion of key signature. This can be challenging since there are many different keys, and the network must learn that the relative positions between notes are important for creating pleasing lines, not the notes' absolute positions. Finally, pieces of music tend to be at least a few minutes long, and it is very difficult for an RNN to learn high level musical structure over a whole song, simply because it needs to look very far back.

Our goal was to create a network that can train on raw musical data from an individual song or corpus of songs and create novel music. That is, we did not want to impose any musical structure on the data, such as parsing the music into measures, telling the network what key the music is in, or labeling chords in the music. This keeps the network general, and in theory it should be able to create music similar to an arbitrary musical genre.

2 Data

We acquired our training data in MIDI format from www.piano-midi.de. The primary song we used was Bach's Sonata No. 23 in F minor, Op. 57, "Appassionata." Details on the MIDI format are discussed below.

2.1 MIDI

We wanted to train the network on MIDI (Musical Instrument Digital Interface) data, since MIDI files are easily available and are the accepted data format in professional digital music production. MIDI conveys musical information through a sequence of event messages such as `NoteOnMessage` and `NoteOffMessage` that describe when a note is being played, at what volume, and when it is released. MIDI uses different tracks that have their own event sequences in order to represent multiple voices or instruments. There are 128 different pitches in MIDI, and MIDI measures time in ticks. The length of a tick depends on the tempo of the song and the resolution of the file. Resolution refers to the number of ticks per quarter note, and tempo tells how long a quarter note is in real time.

We had written a script to transpose songs into the same key, since we believed this would speed up training. We ran into other issues, however. Because every song has a different resolution and tempo, it was difficult for the network to learn quickly. We chose instead to train our models on just one song and have it produce music similar to that one song it was trained on. The network would still be able to compose music indefinitely, and it is general enough that it could be trained on a larger corpus as well.

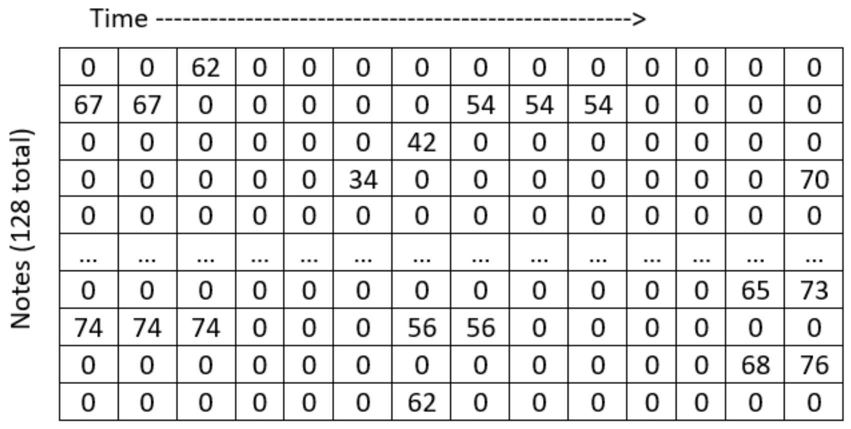
2.2 Data representation

One of the greatest challenges we faced was converting the MIDI event format to a suitable data representation for sequence data. We experimented with several different ways of representing the data for the same LSTM architecture.

2.2.1 Representation 1: Piano-roll matrix with velocity

The first representation we tried was a "piano-roll" matrix representation. There are 128 different pitches in MIDI, so our idea was to convert MIDI files into a two-dimensional matrix representation of dimension $(T, 128)$. At every time step, we sampled what notes were on and at what velocity. So, in the data matrix X, X_{ij} gave the velocity of note j at time step i.

We initially tried making one time step equal to one tick, but the network failed to learn musical structure because the resolution of most MIDI files was too high. Most MIDI files have a resolution of 96 ticks per quarter note, which means that even if we fed the RNN a very long input of length 128, this input wouldn't even have half a measure's worth of musical information - it would be impossible for the network to learn anything meaningful. We decided to make one time step equal to 24 ticks and downsample the notes played at every 24th


Time -->

Notes (128 total)

0	0	62	0	0	0	0	0	0	0	0	0	0	0
67	67	0	0	0	0	0	54	54	54	0	0	0	0
0	0	0	0	0	0	42	0	0	0	0	0	0	0
0	0	0	0	0	34	0	0	0	0	0	0	0	70
0	0	0	0	0	0	0	0	0	0	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...
0	0	0	0	0	0	0	0	0	0	0	0	65	73
74	74	74	0	0	0	56	56	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	68	76
0	0	0	0	0	0	62	0	0	0	0	0	0	0

Figure 1: Visual representation of piano-roll matrix with velocity. Dimensions flipped to appear as MIDI software shows it

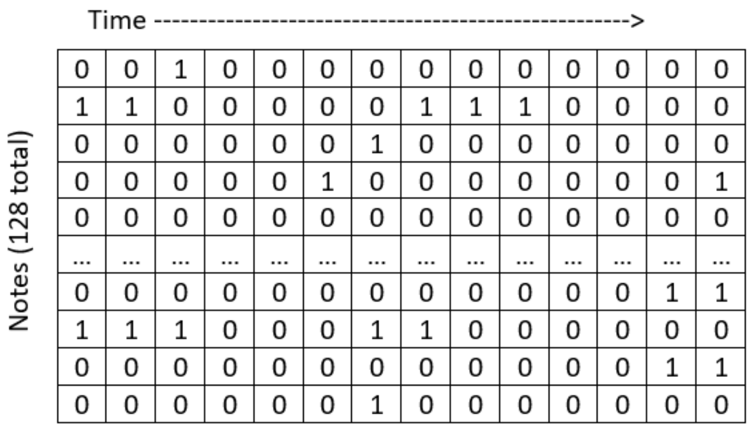
tick for the input matrix. This is equivalent to making a sixteenth note the minimum note duration that the network can learn. This loses some musical information if a song has very fast notes, but this is relatively rare.

In decoding, we must be able to tell if a note is held or rearticulated at a certain time step. If the velocity changed from one non-zero value to a different non-zero value after one time step, we treated this as the note being rearticulated. If the velocity stayed the same over the time step, we treated this as the note being held. A limitation of this representation is that it cannot distinguish between a held note and a rearticulation of a note at the same velocity. This is a rare case, however, and it would have minimal impact even if it did occur.

2.2.2 Representation 2: piano-roll matrix without velocity

The second representation we tried was similar, but it did not convey velocity information. Instead, X_i was a multihot vector where x_{ij} was 1 if the note j was played at time step i at all, and 0 otherwise. The fact that X_i was a multihot vector of probabilities also allowed us to use sigmoid cross entropy loss instead of mean squared error.

In order to distinguish between held and rearticulated notes, we tried including a "hit" matrix of the same dimensions, where the value of position i, j was 1 if note i was articulated at time i and 0 otherwise. This turned out not to be useful, however, so we removed it and chose not to distinguish between held and articulated notes.


Time -->

Notes (128 total)

0	0	1	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	1	1	1	0	0	0	0
0	0	0	0	0	0	1	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...
0	0	0	0	0	0	0	0	0	0	0	0	1	1
1	1	1	0	0	0	1	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	1	1
0	0	0	0	0	0	1	0	0	0	0	0	0	0

Figure 2: Visual representation of piano-roll matrix without velocity. Dimensions flipped to appear as MIDI software shows it

2.2.3 Representation 3: one-hot encoding of note combinations

The third representation we tried was inspired by architectures for language models. Each label was represented as a one-hot distribution over the combinations of notes seen in the training data. From the perspective of language modelling, the prediction vector becomes a distribution over the model's musical vocabulary. This allowed us to turn prediction into a classification problem with a standard softmax cross entropy architecture in the final layer.

2.2.4 Additional input features

For all of these representations, our architecture struggled to learn to hold notes longer than a single time step. To solve this, we found inspiration in the work of Daniel Johnson [3] whose input consisted of 5 different feature types beyond the piano-roll matrix. In our case, we concatenated a length 128 vector h to x where h_i represented how many time steps note i had been held. We hoped that this might encourage the network to learn to hold notes.

3 Implementation [1]

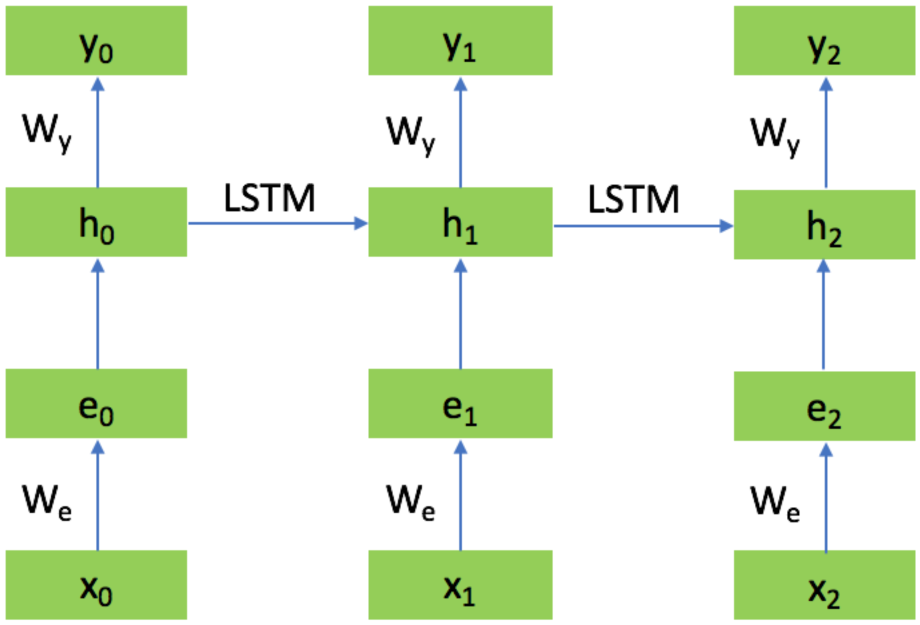
We tried a variety of different architectures based on the data representation, but all were based on long-short term memory networks (LSTMs). We chose

[1] Code available at `https://github.com/jweinstein2/DeepMusic` on branch `working`.

LSTMs because musical lines are heavily affected by what is played in the past, and LSTMs are capable of learning what features are important from the past.

We cut up songs into sequences of length 64 time steps and treated each of these sequences as a training example, where the label is the training example itself. We chose 64 as the time step length because when sampling every 24th tick at a resolution of 96 ticks per quarter note, 64 time steps gives us a training example of length 16 quarter notes, or 4 measures if the song is in the common 4/4 time signature.

3.1 Representation 1 architecture



With this representation, we first used an embedding layer of size 64 that mapped the input to a less sparse representation. Then we had an LSTM layer of size 512, followed by a fully connected layer of size 128. We knew the hidden layer size had to be larger than the input and output size, so the network can store more information. We used a custom activation function similar to ReLU in that the minimum output was 0, but we also set a maximum of 128 (the maximum velocity of a MIDI note). The output of this fully connected layer was the predicted vector at a given time step, and we used mean squared error loss between this prediction vector and the actual vector in the input song. Because we wanted the network to learn velocity in addition to pitch and rhythm so it could learn musical dynamics, the network was not modeling a probability distribution, and mean squared error was our only real choice for the loss function. We used AdamOptimizer with a learning rate of 0.01 and a batch size of 10, and we trained for 200 epochs on one song.

We also tried adding a second LSTM layer of the same size on top of the first, and added dropout layers after both LSTM layers with a dropout rate of 0.5.

Neither of these models were able to produce good results, and we hypothesized that this was due to the choice of data representation and loss function. We discuss this in detail in the qualitative results section.

3.2 Representation 2 architecture

This representation had the same hyperparameters as before, but we changed the loss function and the way the network made predictions. Because the labels were now multihot vectors, we were able to use cross entropy loss after a sigmoid activation in the fully connected layer. We experimented with several different approaches to converting this multihot vector of probabilities into a generated note vector for the next time step.

3.2.1 Deterministic generation

Our simplest approach to generation was to take the length 128 output vector of the fully connected layer o and produce a binary vector $\hat{y}$ for the notes at the next time step according to:

$$\hat{y}_i = \begin{cases} 1, & \text{if } o_i > \epsilon \\ 0, & \text{otherwise} \end{cases}$$

where ϵ is a tunable parameter between 0 and 1. Using $\epsilon = 0.3$, we were able to get pleasing music after training for 200 epochs on a Bach song.

3.2.2 Randomized generation

The second approach to generation introduced an element of randomness. Because the output of the fully connected layer contained the probabilities of each note being played, we used these probabilities directly in producing the multihot prediction vector. The equation was:

$$\hat{y}_i = \begin{cases} 1, & \text{with probability } o_i \\ 0, & \text{with probability } 1 - o_i \end{cases}$$

3.3 Representation 3 architecture

The labels in architecture 3 were one-hot, and we used a softmax layer after the fully connected layer to give us the probabilities of each combination of notes. Thus, the output itself was a probability distribution over the possible combinations of notes.

As with architecture 2, we had both a deterministic and a randomized method to generating the next note from the predicted distribution. The first prediction approach was simply taking the maximum index of the distribution and interpreting this as the prediction for the next time step. The randomized approach was sampling from the distribution according to the probability mass assigned to each class.

4 Results and evaluation

Due to the generative nature of the task, our results come in two forms: qualitative and quantitative. The qualitative results are the nature of the music that each architecture was able to generate. Since this is rather hand-wavy, we also evaluated the different architectures choices in terms of the quantitative metric of perplexity, which is defined for a sequence according to

$$\text{perp}(w_1, .., w_n) = \Big(\prod_i^n p(w_i)\Big)^{-\frac{1}{n}}$$

where $p(w_i)$ is the probability assigned to the correct note combination w_i at time step i.

Roughly speaking, the perplexity of a probability model on a sequence represents how well the sequence is able to be modelled. Low perplexity corresponds to good predictive capability, and high perplexity means that the model is not able to model the sequence particularly well.

One nice property of perplexity as opposed validation loss is that it allows us to directly compare the performance of architecture 2 with architecture 3 even though the loss functions are different. In the case of architecture 3, calculating entropy from a softmax output is trivial since $p(w_i) = \hat{y}_i$.

In the case of architecture 2, we first had to transform our multihot prediction vector into a probability distribution over all the combinations of notes. Let w_i to be a possible subset of notes that could be played at once. Then, we can find $p(w_i)$ in terms of the multihot vector $\hat{y}$ according to

$$p(w_i) = \Big(\prod_{j \in w_i} \hat{y}_j\Big)\Big(\prod_{j \notin w_i} (1 - \hat{y}_j)\Big)$$

Using these $p(w_i)$, we can find the perplexity learning curves for architectures 2 and 3. These perplexity scores allow us to directly compare the ability of each architecture to model a given musical sequence.

4.1 Quantitative results

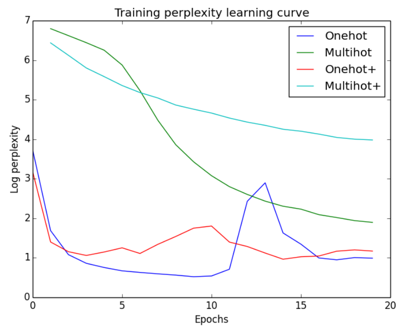


Figure 3: Log sequence perplexity over the training set.

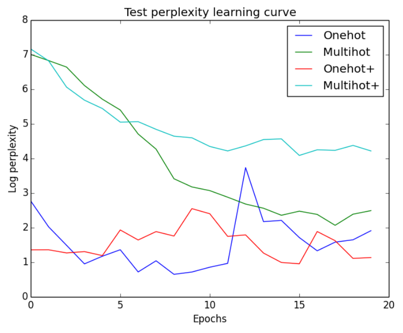


Figure 4: Log sequence perplexity over the test set.

Our quantitative results show a clear difference between the performance of our onehot architecture and our multihot architecture. Our onehot performed better as expected. This matches the qualitative results discussed in the next section. We were surprised to find that adding the additional feature data did not improve our network performance. In the case of multihot it increased our perplexity values meaning worse predictions. In the case of onehot it offered no significant improvement.

4.2 Qualitative results

4.2.1 Representation 1 quality

This architecture was not able to produce much more than constant, quiet, and random noise. We hypothesize that this was due to the fact that we used mean squared error loss between vectors that had a few values around 75 and most values at 0. To minimize mean squared error, the network learned to push most values to something close to 0. Because these values represent velocity in this representation, this resulted in quiet notes of every pitch being played all the time. This motivated us to try data representation 2.

4.2.2 Representation 2 quality

The network appeared to learn some notion of rhythmic structure and beat, and it learned to play pleasing chords and melodies that were in tune. However, it tended not to hold notes for multiple ticks, and it instead played different notes with a lot of silence in between or in quick succession. Augmenting the input features with the hold length vector did not seem to help the network hold notes. There was also little difference between the deterministic and random generation of the prediction vector.

4.2.3 Representation 3 quality

Compared to the previous two models, this network learned the most pleasant note progressions and generated varied note duration and silence in between notes. Qualitatively, we all agreed that this was the best model. Despite its relative success, this network still had major shortcomings. The generation feel into repetitive cycles past the max sequence length from training. The notes were also relatively short and choppy compared to the input. We are still far from human quality composing.

5 Contribution of team members

Every member of our group worked on both data processing and network architecture. In terms of preprocessing, Roland wrote code to convert between MIDI and the piano roll matrix for architectures 1 and 2. Jared generated the onehot encoding for architecture 3. Will debugged the MIDI encoding script and wrote the preprocessing code necessary for perplexity evaluation. For architecture, Will worked on the first-pass implementation of the LSTM, multihot generation, and perplexity calculation. Jared worked on adapting both the architecture and generation methods for onehot prediction and additional features. Roland worked on debugging and improving the LSTM architecture, adapting code to work with multiple LSTM layers, and tuning hyperparameters.

6 Conclusion and discussion

While we were able to get somewhat successful results training an LSTM on MIDI files, the results are not pleasing enough to be compared with human-composed music.

We identify two main areas for improvement that our work was unable to address. First, our network was unable to learn to hold values as long as the input. Secondly, our network was able to improvise infinitely but never modeled the long term structure found in most songs. This failure is expected considering our training inputs were no longer than four measures. Given raw musical information in a similar format to the one used in this paper, it it extremely challenging to learn long term musical structure simply because gradients cannot backpropagate across a whole song. In our literature review, we found no example that successfully learned long term patterns. Lastly, because our training set contained only one song, the network only learned to compose music similar to that one song and did not learn the characteristics that define a genre in general. A much larger corpus of similar music is required to build a model that generates, for example, Beatles-like music, as opposed to music similar to one song from the Beatles.

In order to resolve these problems and improve our network output we propose several changes as the basis for future work. Most simply, we believe our network will have lower perplexity values simply by increasing the depth of our architecture and decreasing the degree of randomness in our output. Although providing the additional hold features did not help our network, different features may work better. For instance, at time-step t_x we could also pass in the previous n time steps: $t_{x-n} \dots t_{x-1}$. This reduces the burden placed on the LSTM nodes to learn short-term temporal dependencies. Finally, because our architecture mirrors the problem of text prediction in NLP, many of the same improvements in NLP can be applied to our architecture. Further research and work is required to selectively apply these improvements.

References

[1] Allen Huang. *Deep Learning for Music.* https://cs224d.stanford.edu/reports/allenh.pdf.

[2] Dan Jurafsky. *Language Modelling.* https://web.stanford.edu/class/cs124/lec/languagemodeling.pdf.

[3] Daniel Johnson. *Composing Music With Recurrent Neural Networks.* Hexahedria, 3 Aug. 2015, http://www.hexahedria.com/2015/08/03/composing-music-with-recurrent-neural-networks/.

[4] Joseph Weel. *RoboMozart: Generating Music Using LSTM Networks Trained Per-Tick on a MIDI Collection with Short Music Segments as Input.* https://esc.fnwi.uva.nl/thesis/centraal/files/f1647680373.pdf.

[5] Li-Chia Yang *et al. MidiNet: A Convolutional Generative Adversarial Network for Symbolic-Domain Music Generation.* https://arxiv.org/pdf/1703.10847.pdf.

[6] Nicolas Boulanger-Lewandowski *et al. Modeling Temporal Dependencies in High-Dimensional Sequences: Application to Polyphonic Music Generation and Transcription.* https://arxiv.org/pdf/1206.6392v1.pdf.