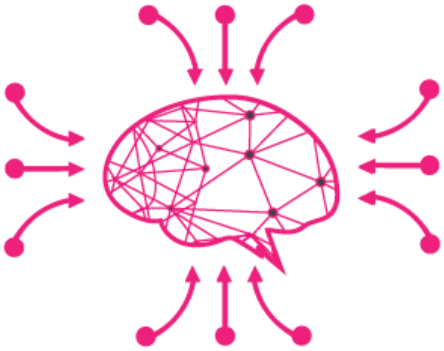
RNN Music Composer

Psych 186B Final Project

Alvin Vuong* & Michael Evans



Introduction

Why Music?

As humans, we like to believe that creativity and intelligence are things that only we can be capable of. However, as technological advancements have shown, machines are becoming increasingly capable of completing tasks previously meant to be uniquely human. Music composition is one of those things. In this project, we raise the question of whether or not a computer program, specifically a neural network (NN), is able to create music? And after this, we also need to consider how its output compares to a human composer, and how the output can even be considered "real"?

The Problem

In order to create what we have proposed, a neural network capable of performing music composition, there are several problems we need to address:

1. A Feed-Forward model (FFNN) is not sufficient. What type of model can we use instead?
2. This will still be supervised learning, but what cost function should we use for our model?
3. How do we convert Musical Instrument Digital Interface (MIDI) data to a usable format for our NN?
4. How do we judge a composition as "good enough"?

Background Research

In order to begin, we had to find sources where sequential data were used; a quick online search resulted in papers with one common characteristic: they all used recurrent neural networks (RNNs) as the underlying structure in order to solve various problems. These included projects done by Facebook and Google where they used RNNs to model speech and video facial recognition in real time (Hak 2014) (Ranzato 2016). Character-level language models also used RNNs (Karpathy 2015). Going through all of these, we eventually came across Daniel Johnson's "Biaxial Long Short-Term Memory Recurrent Neural Network" architecture (Johnson 2017), which was able to handle a song's time and note data very effectively. We used this as our starting point for this project.

Methodology

Setup

For our development environment, we needed to download and install Theano, a numerical computation library for Python with support for neural network frameworks. Explicitly, we followed the

instructions shown here: http://deeplearning.net/software/theano/install_windows.html#install-windows. We ensured we had the following dependencies installed (in the following order!):

- Windows OS 64-bit
- Microsoft Visual Studio 2015
- NVIDIA CUDA drivers and SDK version 8.0: https://developer.nvidia.com/cuda-zone
- Anaconda2 4.3.0 (which includes the following modules)
    - CPython g++ (gcc) compiler http://tdm-gcc.tdragon.net/
    - Python 2.7.13
    - NumPy 1.11.3
    - SciPy 0.18.1
- Additional Python modules installed
    - Theano 0.8.2
    - theano-lstm 0.0.15
    - python-midi 0.2.4
- Microsoft Visual C++ Compiler for Python 2.7: https://www.microsoft.com/en-us/download/details.aspx?id=44266

    After all these are installed, running our `main.py` file in the top directory should work fine.

Dataset: Classical Piano

Okay, so once we have our environment setup, we had to find a good source of data that we could pull from and use. We chose classical piano MIDI files, since they are conveniently formatted in a way that we can easily parse through them. These are located at: http://www.piano-midi.de/midicoll.htm (Kreuger, 2016). These were the same files that Johnson used in his paper.

MIDI Files

All of the files in the dataset are MIDI files. These files are structured in the following way:

- 'song.mid'
    - Tracks
    - Events
        - Time signature
        - Key signature
        - Note
            - **Ticks (time)**
            - **Pitch (frequency)**
            - Velocity (loudness)

We only need the ticks and pitch data from these files.

Parsing a MIDI File

Source: *midi_to_statematrix.py: line 8*

In order to parse through our MIDI files, we find specific NoteEvents, and convert them into binary tuples that represent whether a note is played or articulated (or both) at a certain timestep.

- [1,1] = Played and articulated
- [1,0] = Played but not articulated
- [0,0] = Not played or articulated
- [[0,0], [1,1], [1,0]… [0,0]] (78 possible notes, so 78 tuples)

A part of a song is then represented as a list of these lists of tuples:

[[[0,0], [1,1], … [0,0]], t = 0
[[0,0], [1,1], … [0,0]], t = 1
[[0,0], [1,1], … [0,0]], …, t = 2 ...
[[0,0], [1,1], … [0,0]]] t = N

Input Layer: Binarization of MIDI Data

Source: *data.py: lines 21, 34, 38, 56, 64*

After parsing, we now have to construct a binarized feature set vector in order to feed into our NN. This was done in the following way: (1 timestep = 1 input vector)

- Position [3] → C1 (Index of tuple)
- Pitch Class A, Bb, C, C#, D, D#, etc. (Index of tuple % 12)
  [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0] → C
- Previous Vicinity ± 1 octave in last timestep (played and/or articulated)
  [.., 1, 1, 1, 0, ..] → C4 articulated, D4 played but not articulated
- Previous Context number of each pitch class in last timestep (relative to note)
  [1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0] → Cmaj
- Beat measures split into 1/16ths (reverse binary encoded)
  [0, 1, 0, 1] → 10; 10/16 = 5/8 = 2/4 + ⅛

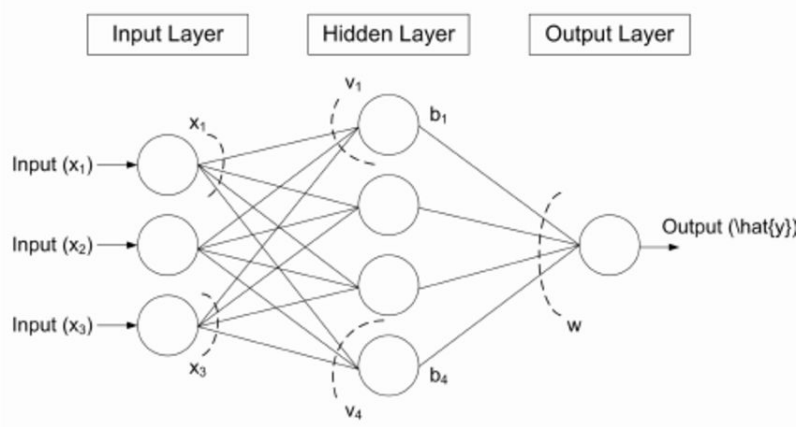
The rationale for these features comes from Johnson's paper as follows:

“

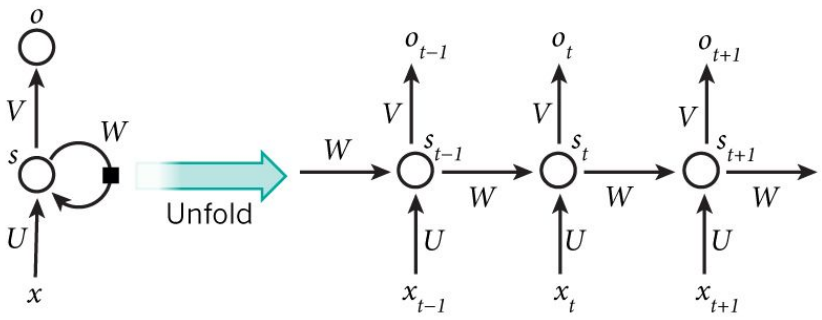
- Position [1]: The MIDI note value of the current note. Used to get a vague idea of how high or low a given note is, to allow for differences (like the concept that lower notes are typically chords, upper notes are typically melody).
- Pitchclass [12]: Will be 1 at the position of the current note, starting at A for 0 and increasing by 1 per half-step, and 0 for all the others. Used to allow selection of more common chords (i.e. it's more common to have a C major chord than an E-flat major chord).
- Previous Vicinity [50]: Gives context for surrounding notes in the last timestep, one octave in each direction. The value at index 2(i+12) is 1 if the note at offset i from current note was played last timestep, and 0 if it was not. The value at 2(i+12) + 1 is 1 if that note was articulated last timestep, and 0 if it was not. (So if you play a note and hold it, first timestep has 1 in both, second has it only in first. If you repeat a note, second will have 1 both times.)
- Previous Context [12]: Value at index i will be the number of times any note x where (x-i-pitchclass) mod 12 was played last timestep. Thus if current note is C and there were 2 E's last timestep, the value at index 4 (since E is 4 half steps above C) would be 2.
- Beat [4]: Essentially a binary representation of position within the measure, assuming 4/4 time. With each row being one of the beat inputs, and each column being a time step. ”

Recurrent Neural Networks

The first thing we had to do was understand how RNNs functioned and how they differed from FFNNs. A 1-hidden layer FFNN consists of a fixed-size input vector multiplied by a weight matrix, which results in the hidden layer, which is then multiplied by another weight matrix in order to get the output vector. This structure can be shown as follows:



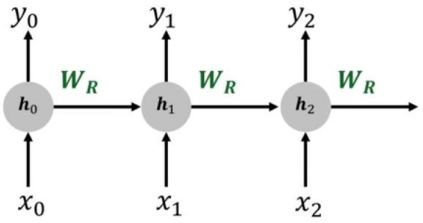
Now, a RNN differs from this in the fact that its hidden layer units also feed into themselves, creating a sort of time axis. This can be shown below:



As you can see above, the time axis can be unfolded to show each timestep of the NN. For RNNs, the states of each recurrent unit depend on the values in the last timestep as well as the new input. This is what sets RNNs apart from traditional FFNNs.

Vanishing/Exploding Gradients

The problem with backpropagation algorithms on RNNs is that when calculating the weight adjustments for each layer, the weight matrix is multiplied T times, where T is the total number of timesteps. With too many timesteps, this can cause what's called a "vanishing" or "exploding" gradient, where the weight adjustment becomes very large (too much correction) or very small (no correction at all), if the weight matrix values are > 1 or < 1 respectively. This prevents us from properly training our network. A visual representation of this phenomenon is shown below:



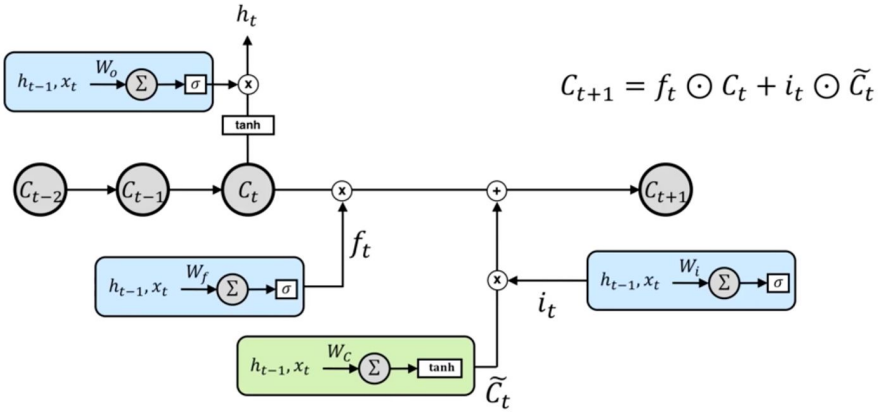
$$\frac{\partial C_{100}}{\partial W_R} = \frac{\partial C_{100}}{\partial y_{100}} \dots W_R \frac{\partial g_{100}}{\partial a_{100}} \dots W_R \frac{\partial g_{99}}{\partial a_{99}} \dots$$

$$\frac{\partial C_T}{\partial W_R} \propto |W_R|^T \left|\frac{\partial g}{\partial a}\right|^T$$

Long Short-Term Memory (LSTM)

The LSTM model for neuronal units allows us to solve the problem of vanishing and exploding gradients by introducing a mechanism for "forgetting" information (Hochreiter, 1997). We add a "forget gate" as well as output and input gates, which all modulate the amount of information we're letting out of the model. Each of these gates have their own weight matrices that are also maintained via the backpropagation model, allowing the NN to decide what to forget and what not to forget.

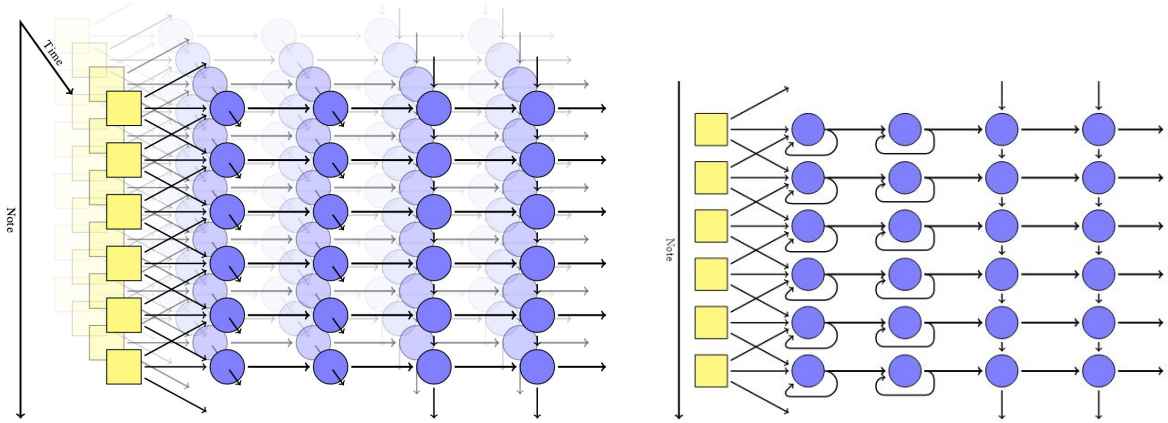
There are 4 gates in total (per neuron) that each take in the activity of the current neuron and multiply by an individual weight matrix, This result is then passed through a nonlinear associator, bounding the final value. A visual representation of this mechanism is as follows:



Hidden Layers: Biaxial NN Architecture

Source: *model.py: line 78*

Johnson's paper describes a novel NN architecture with 4 recurrent layers, called the Biaxial LSTM RNN. The first two layers are recurrent in the time axis, but independent in the note axis. This means that each node feeds into itself but not to the other nodes in the same layer (taking input from its previous state and the previous layer). This allows for these layers to learn the time-based sequential patterns of rhythm, beat, and melody. The other two layers are the opposite: recurrent in note axis, independent in time. This means that each node feeds into the other nodes in the same layer but not itself (taking input from the previous layer and other notes). This allows for these layers to learn the concurrent patterns of chords and harmony. Initialization of this network is handled by Theano.

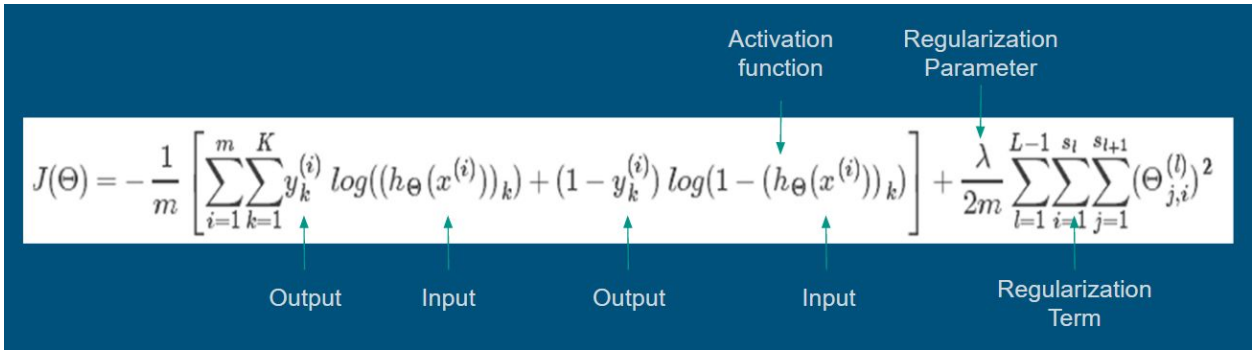


Cross-Entropy Cost Function

Source: *model.py: line 222*

In order to compute our error to be used for training our network, we need to define some sort of cost function. The output of this function will be the value we need to minimize during training. Our error is calculated by taking the binary input vector and comparing it to the probability output vector.

We calculate the according cross-entropy for each timestep, defined below, and we sum up these values for a total aggregate error value.



In our code, we actually used a vectorized form of this equation in order to help reduce the cost of computation for training our network.

Backpropagation

Source: *model.py: line 225*

The cost calculated is then fed into our backpropagation algorithm in order to adjust the weights of our network (W).

We use the ADADELTA (Zeiler, 2012) adaptive learning rate backpropagation algorithm in order to train our network.

Output Layer: MIDI Reconstruction

Source: *model.py: lines 253-258, midi_to_statematrix.py: line 90*

For each note in a given timestep, we output a play and articulation probability. Articulation here means that a note is “hit” during this time. For this reason, if we have a note not being played, it can’t be

articulated. If a note is played but not articulated, then it will be held. This can only occur if the note had been played on the last timestep.

We check whether these values determine whether a note is played or articulated by using a uniform-distribution random number generator and comparing the value to the corresponding probability threshold. If the random number is less than the play probability then it will be played, similarly if a second random number is less than the articulate probability then it will be articulated (this is only checked when the note is already determined to be played). Once our model is created and trained, there should be only high probabilities for notes that make melodic and harmonic sense with the previous timestep.

Results

Output Compositions

We have included a folder of samples, named REPORT_SAMPLES, with the following naming convention: *'h1-h2-h3-h4-05-i.mid'*, where the first four numbers, h1 to h4 are the number of units for each hidden layer of the neural network, and the last number is the number of training iterations after which the model was tested to produce that specific sample MIDI output.

During training, we also printed the model's error amount for that epoch to see if the quality of the song samples varied systematically with the error. In general, a higher error value meant a lower quality song. Lower quality here means that the song sample did not sound human-made (strange timing, non-matching chords/melodies). When error was at its largest, we observed recordings of almost every note played at every timestep, resulting in a distorted, unpleasant white noise. As the error decreased, melodies and a consistent beat could be heard increasingly so. We also noticed that lower error didn't necessarily mean a good sample – though samples with smaller errors tended to sound more pleasant.

We noticed that as the number of iterations increased, the error decreased on average. This makes sense, as our model is designed to decrease the error.

We also varied the size of our models to see differences in sample quality. We noticed that as the size of the neural network increases, the number of training iterations it takes for our model to converge on a minimum decreases. So, between two different-sized models with the same number of training iterations – the one with more neurons tended to sound more human-made on average.

Turing Test

Because there is no way to truly measure the quality of our songs (since it's subjective), we decided to run a Turing test on a small sample of 20 of our friends.

We asked each of them to pick out the actual human-created piece out of two audio clips. These two audio clips included our "best" 16-second sample from our network as well as an actual 16-second clip from a piece by classical composer Isaac Albéniz. "Best" here means that Michael and I deemed it to be the most organic and human-like out of all the other samples. Our artificial piece would be deemed successful if participants could not tell the difference between the human-made and the NN-generated samples, and the selection would be no different than as if people chose them at random chance: 50%-50%. We found that 11 out of 20 subjects (55%) chose the NN-generated sample to be the "human-created" sample, showing that our NN, at its best sounds no different from an actual human composer.

In order to test the overall performance of our model, we also asked each subject to classify whether or not a sample was human-made or NN-made for a batch of 5 randomly selected samples, for a total of 100 samples. Every sample was generated by our NN with a different set of input songs. What we found was that 34 out of the 100 randomly played samples were categorized as "human-created." This means that our NN can trick a person into thinking its compositions are human created 34% of the time.

Problems with the Model

One of the problems we noticed about our model was that for any simulations for longer than approximately 40 seconds, our model would deteriorate and only either play a single repeated note over and over again ad infinitum or it stayed silent. We believe that this is the result of the network getting caught at a local minimum for error and the best option is to play nothing or a single note and time, thereby minimizing the error.

Too low of an error value also caused the model to completely replicate songs and there would be no "creativity" by the model. This may not necessarily be a failure, but it's not the purpose of our model which is to create novel and unique music. We found that there is no real good "error threshold" by which we could discern "good" pieces from "bad" ones. Sometimes high error values could still produce unique and interesting music samples. However, there seemed to be an amount of error that produced "better-sounding" samples at a higher frequency than other error rates. But this amount of error varied upon changing input songs and the dimensions of our NN.

Conclusion

---

https://news.developer.nvidia.com/ai-composer-creates-music-for-films-and-games/

A quick look at the above link, shows the state of the art and potential in this field. AIVA Technologies used CUDA, TITAN X GPUs, and TensorFlow to create an entire symphonic composition all done using neural networks. Once this technology becomes more and more mainstream, we will see an increasing number of NN-generated music for games, film, and other entertainment. The applications for RNNs are immense and the implications of more complex NN architectures being able to replicate and

surpass human ability are far-reaching. We will soon have artificial technologies capable of producing art, writing books, and holding conversation.

Works Cited

- Hochreiter, S., Schmidhuber, J. (1997). “Long Short-Term Memory.” Neural Computation. Vol. 9 No. 8: 1735-1780. http://dl.acm.org/citation.cfm?id=1246450

- Johnson, Daniel (2017). “Generating Polyphonic Music Using Tied Parallel Networks.” EvoMusArt 2017. http://www.hexahedria.com/files/2017generatingpolyphonic.pdf

- Kreuger, Bernd (2016). “Classical Piano MIDI Page.” http://www.piano-midi.de/midicoll.htm

- Nervana (2016). “Recurrent Neural Networks.” https://www.youtube.com/watch?v=Ukgii7Yd_cU

- Ranzato, M., Chopra, S., Auli, M., Zaremba, W. (2016). “Sequence Level Training with Recurrent Neural Networks.” ICLR 2016. https://research.fb.com/wp-content/uploads/2016/11/sequence_level_training_with_recurrent_neural_networks.pdf

- Sak, H., Senior, A., Beaufays, F. (2014). “Long Short-Term Memory Recurrent Neural Network Architectures for Large Scale Acoustic Modeling.” Interspeech 2014. http://193.6.4.39/~czap/letoltes/IS14/IS2014/PDF/AUTHOR/IS141304.PDF

- Zeiler, M.D. (2012). “ADADELTA: An Adaptive Learning Rate Method.” https://arxiv.org/abs/1212.5701