Bach to Basics: A Simple Evolutionary Approach To Music Composition

Anonymous Author(s)
Affiliation
Address
`email`

Abstract

This project attempts to use simple machine learning techniques, specifically a two-layered neural network, to generate new songs that are subjectively similar to a set of input songs, but are not exactly the same. A number of different encodings for songs were used, to varying degrees of success.

1 Introduction

Generating music through the use of computers can range from simply augmenting sounds that are recorded by artists to completely generating and performing new pieces of music. The 1950's saw the first composers begin to leverage the power of using computers for composition, but many of these composed pieces did not sound at all similar to conventional music. In [1], Levitin devotes an additional attention to the fact that when humans listen to music, they form an expectation of what the next part of the song should sound like based on what they have heard before in other songs, or even previously in the current song. Many of the initial computer compositions are so completely different from what people are used to hearing that it can take time to build a desire to listen to more of them.

One way to get around the concern of having compositions that are drastically different is to involve humans in the process of composition; [2] is one such method using genetic algorithms that require humans to evaluate a fitness function. [3] uses recurrent networks that have been trained on different sets of music to generate compositions, with varying degrees of success, that are similar to those it has been trained on. One of the potential disadvantages of having a computer generate music is that it is often lacking in 'expression', or the subtle changes in timing/emphasis on notes that all human performers add to their pieces. [4] proposes a method of using an agent based approach to addressing this issue. Multiple agents are all given the same base melody to play, but each of them have their own preferences on how the piece should be played with respect to things such as 'note punctuation' or 'loudness emphasis'. The agents then perform their piece to all of the other agents in the group, who in turn evaluate each performance. If an agent encounters a performance that fits better with its own preferences, it updates its own performance to become closer to that of the performance that it liked.

In this project, the system demonstrated by [4] was adapted to compose new pieces of music given a number of samples that a user finds interesting. This method was applied to a number of different computer music files, namely sample based files and midi files, with the hopes of creating a piece of music that sounds subjectively similar to the input sets, but is unique from them.

2 Method

As mentioned previously, when a listener is presented with a piece of music, they are constantly predicting what the next part of the song will sound like given their past musical experiences. When

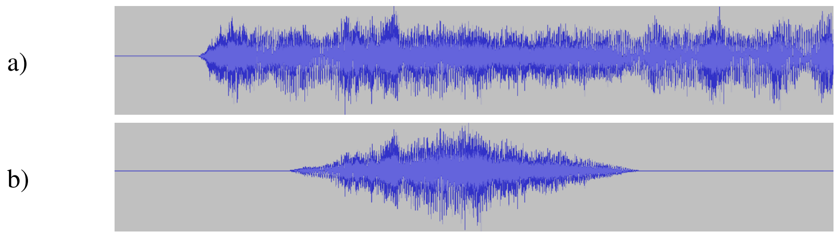

Figure 1: Windowing applied to an audio signal. a) shows the signal before windowing. b) shows the result after a windowing function was applied

the music deviates from expectations, listeners are forced to update their rules for prediction. This is the same paradigm that was implemented to create music in this project. Regardless of the data used, the system works by taking a number of input songs that a user likes, and aims to have something similar generated from them. After selection of these song files, the system proceeds as follows:

1. For each song in the data set, train a model that can predict what is about to happen next given a small window of the song that happened previously. For example, if the model is shown what happens between times 3 and 4 seconds (giving a one second widow), it will predict what happens at time 4 seconds.
2. Have the model 'perform' the song, by sliding the window from time 0 until the end of the song it was initially trained upon, at each step recording what the model predicts.
3. For each model, loop over all the generated performances, updating the model each time to produce a song that is closer to this performance.
4. Repeat at step 2, using the updated models.

This mirrors the design of [4], except that here models that capture a song are used in place of agents that capture musical expression. However, there is no fitness criterion that steers which performances the models are trying to adapt to. Since the user has selected songs that they feel are all 'good', features from all the provided examples are worth having the models attempt to capture them. A two layered neural network was chosen as the model that would be used for performances, similar in design to the one implemented in Homework 5 to perform classification on the MNIST data set. All of the input nodes are fed into a series of $\tanh$ activated hidden nodes, which are then connected to the output nodes. Whenever a model is trained on a specific song, gradient descent is used to tune the weights with respect to an error function that changes depending on the data set. Furthermore, the number of nodes at each layer of the model are chosen specifically for each data set, as described in the following sections.

3 Data set 1: Sample based audio files

Sample based audio files are the most common form of storing music used today - they simply discretize a continuous analog signal that represents the sound waves that need to be transmitted to the user. However, these files require a huge number of data points to represent the high frequencies that humans are capable of hearing, most often 44100 samples a second. To make the signal more manageable, it can be broken up into smaller pieces using a windowing function - producing a smaller signal that represents the entire song around the point which the window is centred (see Figure 1). By giving the model a number of consecutive windows into the past of a song, it should predict what the next window should look like. This poses a problem however, since each of the samples in a window cannot change very much from their neighbours; should this be the case the ear will detect these abrupt changes as clicks in the audio. As such, if a model that is used to predict the value of future samples from previous ones is off by even a minor amount, a listener will be subjected to a decidedly unpleasant sound, one that is not found in any form of music. To get around this problem, the audio signal first needs to be transformed into the frequency domain.

By taking the Fourier transform of the windowed partitions of the audio signal, an audio spectrum can be produced for a song. [5] describes in detail this process while Figure 2 a) shows an example of such a spectrum. Since most songs involve holding a note for some interval of time that can be measured by the brain, using a Hann windowing function that is 0.2 seconds wide, sampled at 10Hz allows for a song to be reconstructed from its spectrum, with negligible loss in perceived quality. This means that the model is now given a number of vectors representing the frequencies present in the previous windows, and can predict what frequencies it expects to find in the next window. An advantage of using this method lies in the reconstruction of the audio signal by performing the inverse Fourier transform. Discontinuities in the frequency domain will be smoothed out when converting it back into an audio signal, preventing any loud clicks or pops in the generated song.

In these experiments, a Hann window, as mentioned previously, of 0.2 seconds sampled at 10Hz, was used to compute the audio spectrum. At a 44100Hz sample rate for audio, this produces a vector that consists of 4000 complex elements when the maximum frequency is limited to 10KHz. While humans have a hearing range of approximately 20-20KHz, songs rarely make heavy use of high frequencies, and so a compromise was made to keep the model size manageable. These are interpreted as 8000 real numbers by the network. The model uses the previous second as input values (10×8000 input nodes are used), from which it must predict the next piece of the spectrum (1×8000 output nodes), with 150 hidden nodes. This means that the model has roughly 13 million parameters in the form of weights between nodes. In order to make training practical, the algorithm was implemented with Theano [6] to exploit GPU parallelism.

When performing gradient descent, the following cost function was used:

$$c = \|\hat{\mathbf{y}} - \mathbf{y}\|_2 + \big|\, \|\hat{\mathbf{y}}\|_1 - \|\mathbf{y}\|_1 \,\big|$$

Where $\hat{\mathbf{y}}$ was the predicted vector of values, and $\mathbf{y}$ is the expected vector of values. This was chosen to penalize the network for producing the wrong values when compared with the expected (the 2 norm), as well as ensuring that differences between the 1 norms of $\hat{\mathbf{y}}$ and $\mathbf{y}$ are minimized. The extra step is added to help guide the network to create predictions that have roughly the same intensity as the expected data - this turns out to make a huge difference in practice.

Results

The networks were trained on sets of four 30 second audio clips, with all the clips from each set from the same composer and having similar styles. Initial training took approximately 20 minutes on a Nvidia 540M GPU, consisting of 12000 updates of gradient descent with a learning factor of 0.01. At this stage, all of the networks were able to successfully predict a spectrum that sounded very similar to what they were trained upon. The embedded files `viv3.mp3` and `viv3.0.mp3` show an example of one such input file, and the performance output respectively. After the training was done, the algorithm proceeded as in Section 2, applying 1000 updates of gradient descent to each model on every other performance. After six iterations, the performances of all the networks became very similar to the one found in `viv3.6.mp3` - they no longer sounded like music, rather they were slightly less then random noise. A comparison of three spectrums can be seen in Figure 2.

These types of results were observed no matter which style of music was used for the initial audio clips. Sets of rock, classical, electronic and jazz clips all produced songs that lost all of their structure after a few iterations. It was believed that the frequency domain was too complex to be grasped by the simple network, in part due to its massive size and the harmonic fundamentals that have a complex relationship with one another. In the hopes of getting around these perceived barriers, a simpler representation of music was used.

4 Data set 2: Midi files

Midi files were one of the first formats used for music composition. They contain a list of messages that specify when a note should start, and when the note should stop in a song. The notes fit into an integer range that fall into the chromatic scale. Midi synthesizers are then responsible for taking these messages and producing an audio signal that can then be listened to. Because of this, midi files are limited in what kind of music they can capture, and therefore should be easier for a simple model to capture properly.

To make matters even less complex, the only types of midi files used contained a single line of melody, so only one note was on at any point in time. Four different types of encodings were used a)

b)

c)

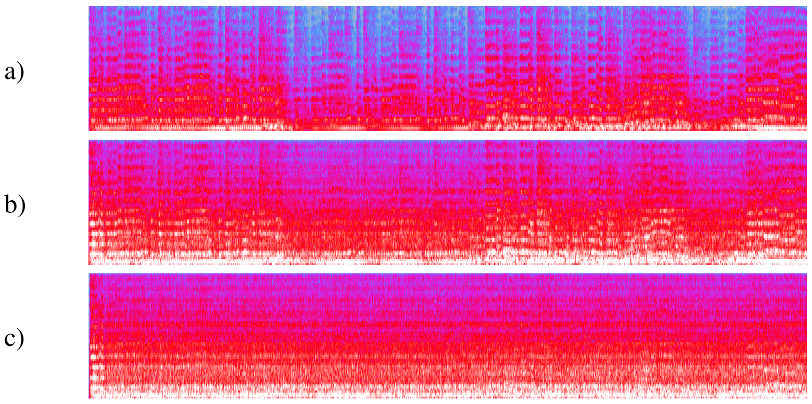

Figure 2: Power spectra of different songs. The y-axis corresponds to a frequency, while the x-axis corresponds to time. Values closer to white are louder, while blue values are quieter. a) is the spectrum of the original piece of music, in this case, Vivaldi's *Spring III: Allegro* (`viv3.mp3` in the embedded files). b) shows the power spectrum of the song produced after the network has been trained on the above piece. Although it is not a perfect match, it can be seen that it has more intense areas in roughly the same locations as the original song (`viv3.0.mp3` in the embedded files). c) shows the power spectrum after six iterations of the method. Note there are almost no obviously similar features to the previous two spectra (`viv3.6.mp3` in the embedded files).

to represent a midi file, the first two of which discretized the song into small time steps, while the latter two operated on the note level.

Encoding 1: Absolute discretization

Similar to the sample based method described in Section 3, the midi song is sampled at fixed intervals through time, each sample holds the integer corresponding to what note is currently being played. In the case that no note is currently on, the value 0 is recorded. In the experiments for this encoding, the song was sampled at 32Hz. The model took as input the previous second of song (corresponding to 32 integers), and was responsible for producing 255 probabilities that would indicate the value of the next note. The network used here was a perceptron, with the output nodes being fed into a soft-max function to produce probabilities for each output, exactly as done in the aforementioned Homework 5. The cost function is the negative log likelihood of an output value occurring. A total of 60 hidden nodes were used, giving the network approximately 17 thousand parameters.

Encoding 2: Relative discretization

This encoding is nearly identical to the previous method, except that instead of encoding the value of the note at a particular time step, the relative difference in notes was stored. For example, if a middle C is played (midi value 60), held for one second, then followed by an A (midi value 69), the system only stores a value of 9 after 32 zeros from where the C began. The network uses all the same parameters as above.

Encoding 3: Absolute note value

Rather then sampling the song at time intervals, each note is encoded as it's midi value along with its duration in seconds. In these experiments, the model is given the previous 8 note encodings, and attempts to predict the two values - the midi note number and the duration of the next note. This means that there are 16 input nodes and 2 outputs, with 60 hidden nodes giving the model approximately one thousand parameters. Here, the cost function is just the 2-norm of the difference between the prediction and the expected value.

Encoding 4: Relative note value

The final encoding scheme is similar to the previous one, except instead of encoding the absolute value of the midi note, the relative difference between changes is stored instead. For example, if the encoding of two notes in a song under encoding 3 looked like $[(60, 1.0s), (51, 0.5s)]$, under this encoding scheme the second note would be encoded as a $(-9, 0.5)$. All other parameters to the network are the same as in encoding 3.

Results

For each different encoding, results varied widely. In each case, the network was initially trained for 12000 updates of gradient descent for the initial performance (generally taking under a few minutes to train), while subsequent updates to the model used 1000 steps for each performance. There were six different midi files in the set used - Bach's *Toccata and Fugue in D Minor*, Beethoven's *Fur Elise*, Bach's *Minuet in G*, Tchaikovsky's *Dance of the Sugar Plum Fairy*, Mozart's *Eine kleine Nachtmusik*, and Rossini's *William Tell Overture*.

Encoding 1: Absolute discretization

Using the absolute method of discretizing the midi song, the network struggled to reproduce the given input song. Most notably, it often produced a series of notes that all had the same tone, but of a very short duration repeated one after another. It also generated sequences that oscillated between two notes, often around the point where a song transitioned from one point to another. Even by extending the initial training period, these artifacts do not disappear. Their presence causes the generated pieces of music to be musically uninteresting, as they often involve repeating the same note over and over for a long period of time, with the occasional jump to a new note randomly throughout.

Encoding 2: Relative discretization

Under this encoding scheme, the networks often fail to produce any output after initial training. Since this encoding has very few non-zero values, the network will demonstrate a very high probability that the next output is always zero, regardless of the input. As such, subsequent training fails to produce any music, since the zero-filled performances just reinforce the network to output a zero with higher probability.

Encoding 3: Absolute note value

When the model is responsible for predicting the absolute value of the next note, it is able to produce a performance that is similar to the input after initial training. However, when subsequent updates are run, the performance does not change significantly between iterations. It is believed that this is due to the fact that the absolute values are encoded in the notes - if another song has a pattern that occurs when all the notes are transposed by a constant number of 12, this type of encoding would cause the network to be forced to learn a new feature for both the original pattern and the transposition. As such, subjectively the author notes that the performances of later iterations of the networks do not vary significantly from the initial performance.

Encoding 4: Relative note value

In the final encoding scheme, the initial performance was nearly the same as the performance using encoding 3 - except that when this performance makes a mistake, the rest of the performance is also transposed by the particular amount that the network made a mistake from. This can be seen in Figure 3. The songs produced are different from their originals, but they still remain partially melodic, but are not something the author would wish to listen to repeatedly.

5 Future Work

The most clear next step for this system would be to increase the complexity of the model, perhaps allowing it to capture the complex structure of the inputs with more accuracy. In addition, the parameters such as the input window size, initial training time, and learning rate were all chosen somewhat arbitrarily. Applying an algorithm tuning process such as [7] could be used to select parameters that try to optimize the output performances with respect to some metric. Furthermore, more complex optimization methods or different cost functions could be used that would take into account the musical quality of the difference. For example, penalizing a prediction that is a semi-tone away from the actual value more then it would penalize a prediction that is off by just a single tone. Hopefully, using these two last methods would prevent the need for moving to a more complex model, allowing for simple methods to generate music.

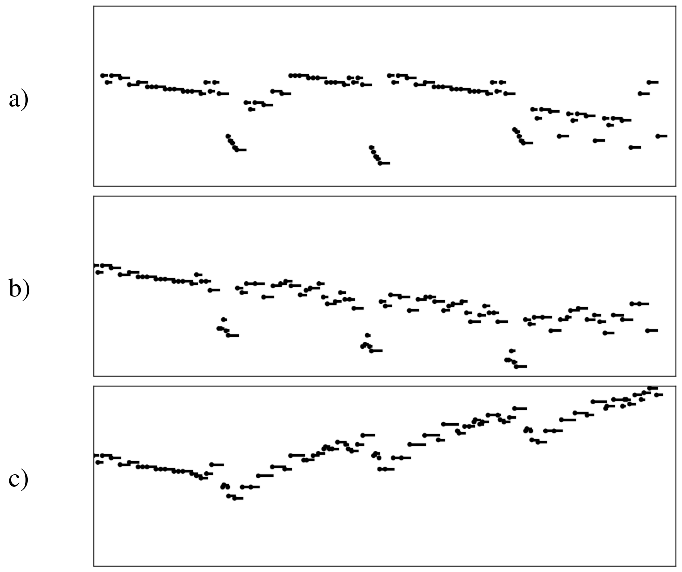

Figure 3: Midi representation of songs. The y-axis corresponds to note's midi value, while the x-axis corresponds to time. a) is the midi file of the original piece of music, *Dance of the Sugar Plum Fairy* (`sugar.mid` in the embedded files). b) shows the midi file produced after the network has been trained on the above piece. You can see that the general structure of the song is reproduced, but the values of the midi notes wander a little bit (`sugar.0.mid` in the embedded files). c) shows the midi file after six iterations of the method. (`sugar.6.mid` in the embedded files).

6 Conclusion

Although no method of encoding music tested throughout this project produced music that was subjectively satisfying to listen to, the fact that the simple two-layer network was able to often able predict what the next part of a song should sound like is encouraging. Although gradient descent was able to support this, it failed to achieve the type of learning required to get the networks to produce different kinds of songs that are similar enough to their inputs that they would be something a listener would find appealing. Further work is required to achieve such a goal, with some possible avenues of exploration outlined in Section 5.

References

[1] D.J. Levitin. *This is your brain on music: The science of a human obsession*. Dutton Adult, 2006.

[2] http://evolectronica.com, November 2011.

[3] C.M. Michael. Neural network music composition by prediction: Exploring the benefits of psychoacoustic constraints and multi-scale processing. *Connection Science*, 6(2-3):247–280, 1994.

[4] E.R. Miranda, A. Kirke, and Q. Zhang. Artificial evolution of expressive performance of music: An imitative multi-agent systems approach. *Computer Music Journal*, 34(1):80–96, 2010.

[5] J.O. Smith. *Spectral Audio Signal Processing, 2011 Draft*. http://ccrma.stanford.edu/ jos/sasp/, accessed November 12, 2011. online book.

[6] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2010. Oral Presentation.

[7] F. Hutter, H.H. Hoos, K. Leyton-Brown, and T. Stützle. ParamILS: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36(1):267–306, 2009.