A Neural Network-Based Chant Melody Composer

Survey of Artificial Intelligence — Anthony Kozar
Spring 2002 — Project Report

"Current research suggests that with enough melodies a net might, for example, replicate new works in the style of learned music."
– David Cope (Cope 16).

Introduction

The goal of this project was to implement a melody composing system using a neural network. Particularly, after basic testing of the network implementation code, the network was trained using existing melodies from the body of traditional Gregorian chant. Gregorian chant was the primary sacred music of the Roman Catholic church for many centuries – being widely adopted around the ninth century and continuing to be used in one form or another through the twentieth century (Hiley, 479). The style of this music has often been difficult to analyze using traditional methods but it is clear that it contains many signature patterns and note groupings (Hiley, 46-7). An underlying premise of the project was that a neural network would be capable of learning some of the local patterns occurring in Gregorian chant from a series of examples. The trained network could then be used to "predict" the next note in a melody given some context taken from the melody up to that point, and through iteration would be capable of producing new melodies containing some of the stylistic signatures of the repertoire used to train the network.

Once a neural network is trained, it becomes a static computing machine; a network will always give the same output for a given input because it does not retain any internal state. Because the network is being used to compose the notes of a melody sequentially and one at a time, some mechanism had to be devised to allow the network to "remember" the context it is working within. The basic principle employed to allow the network to retain some state is called markov chaining. A markov chain is a sequence in which the value of each new member depends on the values of some number of previous members of the sequence. Thus, by including several of the previously generated output notes as inputs to the network, the next note in a melody could be determined based on the pattern of notes immediately preceding it. The process starts by presenting the network with a single starting note and feeding back successive outputs to the inputs. This iterative process would produce a sequence of notes which is the new melody.

To connect this to a neural net, I will use n input groups (each group consisting of several neurons representing different notes) with one group being the current note of a melody and the others the previous n-1 notes of the same melody. A single output group will constitute the output layer of the net. I plan to use unsupervised learning employing Hebb's law. I will probably experiment with many different internal structures and starting configurations as well as vary the number of input groups in order to achieve the most pleasing results.

Implementation

It was decided early on to implement the melody composer in the most popular language for artificial intelligence research, Lisp. In particular, Macintosh Common Lisp – a modern

implementation of the Common Lisp standard – was chosen for several reasons. The interactive environment of MCL makes it possible to quickly prototype and test variations on code without sacrificing too much execution speed since it is compiled. Also, a free and open source framework for music programming called Common Music is available in Lisp code running within the MCL environment. Using Common Music made it possible to get fast and audible feedback about the musicality of the network's output by playing the melodies as MIDI sequences[1]. The third reason for choosing MCL was because it was available on my home computer.

The code for the melody composing system is divided up into several conceptual layers which are portrayed in figure 1. Because there is no standard matrix library in Common Lisp, I wrote one (see code listing 1) to simplify the calculations within the neural network. My neural network library (code listing 2) is built on this matrix library and implements a class for creating networks that use the Hebbian learning algorithm. Hebbian learning was chosen because it is a form of unsupervised learning (which I felt would be an asset in possibly allowing the network to be more "creative") and because it was rather straightforward to implement (Negnevitsky 198). The `hebbian-net` class is generic in the sense that it allows any number of layers with any number of neurons in each layer. It is also possible to select the activation function, learning rate, and forgetting rate for each net object created (Negnevitsky 199). Three methods are provided for initializing the weight and threshold matrices: `hnet-initialize-random`, `hnet-initialize-identity`, and `hnet-initialize-limited-identity` (a variation I tried while testing the melody composer). Finally, since the structure of the network is no different than nets which use other learning algorithms such as back-propagation, it is possible to substitute another learning function for the class method `hnet-train` if one wanted to use something other than hebbian learning.

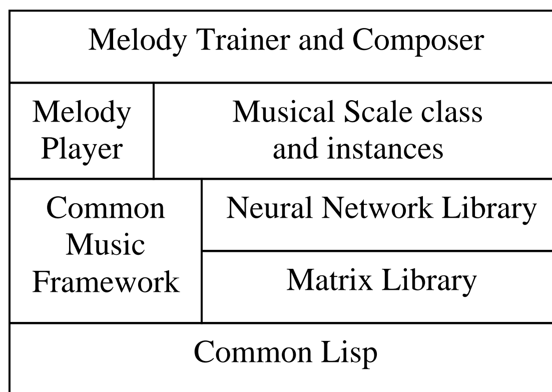

Figure 1. The layered structure of the melody composer system.

The musical scale layer implements a class (see code listings 4 and 5) that allows for the encoding of the notes of a scale as numbers which can eventually be presented to a neural net. Also included are the definitions as global variables of the scales which I used in testing the system. Another very small layer (just one Lisp form) is shown in code listing 7 and defines a

[1] MIDI stands for Musical Instrument Digital Interface and is a standard for storing and playing musical data comprised of lists of notes and other musical "gestures" instead of using direct audio recording and playback.

process object (part of Common Music) called `chant-player` which transforms a list of notes into a MIDI sequence and directs the computer's MIDI implementation to play the sequence. This is how I listened to the results of the system (as well as tested the input for correctness).

The uppermost layer and the most abstract is the layer for training networks with melodies and composing new ones. This layer effectively insulates the user of the system from worrying about how the neural network is implemented and used by the melody composer. The function `train` takes as arguments a `musical-scale` object, a list of melodies (each a list of notes), the markov order of the composing process (i.e. how many previous notes are used as inputs), and the number of epochs for the training. `train` takes care of creating a new neural net object of an appropriate size given the scale and markov order and makes calls to `train-melody` repeatedly for each of the melodies it was passed. The last important function in this layer is the generate-melody function. It takes a previously trained net, a scale, a starting note, and a length (in notes) and creates a new melody represented as a list of note names within the given scale.

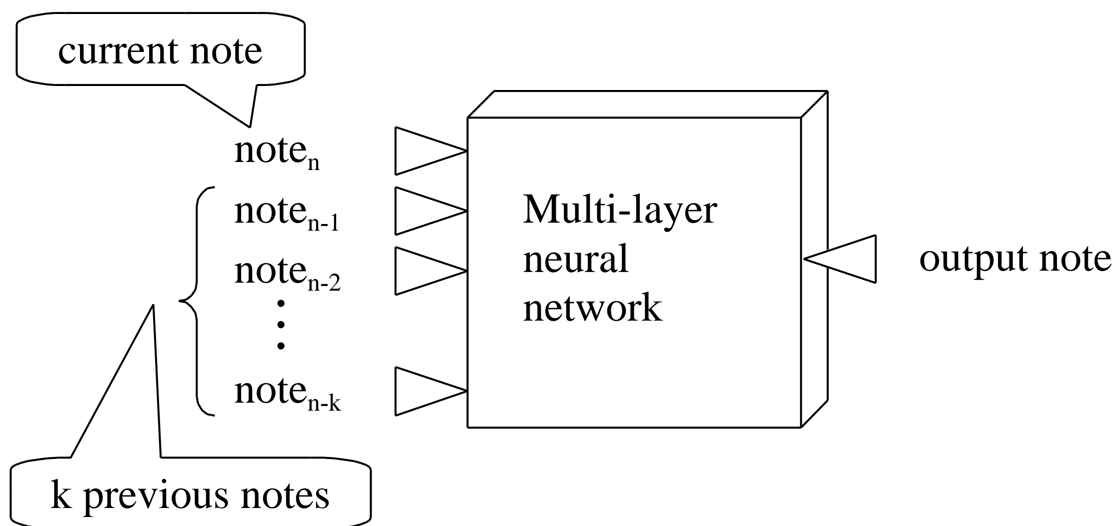

Figure 2. The structure of the melody composer.

The basic structure of the melody composer is shown in figure 2. During training, the current note of a melody and the k previous notes (k is the "markov order") are presented as input to the network. On the next iteration, the current note becomes the next note in the melody and all of the previous notes shift accordingly (down in the diagram). The output note is ignored during training (since this is unsupervised learning). After training, when the net is being used to compose a new melody, the current note is left empty (see below) and only the previous notes are given as input. The output note is taken as the next note in the sequence and is fed back to the position of note n-1 for the next iteration while all the other previous notes shift down one slot in the input.

Each input and output note in figure 2 actually corresponds to a group of neurons within the input or output layer, respectively, of the network. These groups have one node for each pitch or other musical gesture represented by the current `musical-scale` object being used. For example, the scale labeled `*c-major-scale*` in code listing 4 is defined to have eight pitches and so each input and output group in a net using this scale will have eight neurons. Input to a note group is a vector of binary digits with a '1' meaning that a particular note is 'on'

and a ‘0’ indicating that it is ‘off.’ Thus, for each input group, there is only ever at most one node being activated during each iteration of training or composing. An empty note is represented by a vector of all zeros. Empty notes are presented as inputs to the current note group during composition and to previous note groups when there are not enough previous notes to fill all of the slots (such as at the beginning of training or composition).

Early Testing and Results

Before attempting to use the melody composer to imitate chant melodies, I first tested the code with smaller and simpler examples using a pentatonic scale (a five note scale – see the definition in code listing 4). As one test, I trained a network with the following command:

```
(train *pentatonic-scale* '((c e g d c e g a g)) 4 10)
```

This creates a new network using the pentatonic scale and a markov order of 4 and then trains it for 10 epochs on the melody given as the second argument. The network has two layers with 25 nodes in the input layer and 5 in the output layer. The network was initialized with the `hnet-initialize-identity` method which assigns weights of 1.0 to connections of the current input note group to the output group for nodes which represent the same pitch. All other weights are set to zero and the thresholds are randomized. I tested the trained net by giving it individual inputs starting with only one previous note, C. Table 1 shows the outputs for several inputs to this network.

Input	Output
(nil c nil nil nil)	(E)
(nil e c nil nil)	(G)
(nil g e c nil)	(D A)
(nil d g e c)	(C)
(nil a g e c)	(C)

Table 1. Several inputs and outputs to a simple network.

The nil values are empty notes and the first slot of each input which is always nil corresponds to the current note group in figure 2. So when the net is presented with only the starting note C (row 1 of the table), it produces the output note E, which matches the original melody the net was trained with. Taking E as the next note of a melody and substituting it back in for another iteration gives the input pattern in row 2 which produces the output G. Since G always follows the pattern C E in the training melody, this can also be considered a match. Row 3 of the table presents C E G as the previous notes and obtains two notes as outputs. When this happens, it is assumed that the network learned both patterns and that either note is a possible next note for the melody being generated. The `generate-melody` algorithm deals with this situation by randomly selecting the next melody note from among all of the output notes. In this case, it is clear that the network has learned the patterns C E G D and C E G A from the training melody. The last two rows of the table show the output of the network when either D or A is taken as the fourth note of the new melody. The output of C for row 4 is completely expected while the output of C for row 5 shows that the net did not learn the pattern of the last five notes of the input melody. When the system is asked to generate a complete melody from this example, it generates a random sequence of the two patterns C E G D and C E G A. While the goal of the melody composer is not to exactly duplicate its input, I took these tests as an indication that the implementation was working – that it was indeed learning note patterns from its training examples.

Problems started to appear, however, when I expanded this simple example to a longer training melody. Training with the following statement

```
(train *pentatonic-scale*
      '((c e g d c e g a g e d e d c g e g a g)) 4 10))
```

produced a network which did not behave as expected. The summary of outputs for various starting notes shown in Table 2 is very telling. It shows for every possible starting note except A, the trained net returns all possible outputs. And starting note A returns all notes except G as output. These examples are pretty typical of the output of the network regardless of its input. What they indicate is that at each iteration of generating a new melody, the selection of the next note is pretty much random from among all of the notes in the pentatonic scale. This clearly will not reproduce any of the patterns in the training melody except accidentally.

Input	Output
(nil c nil nil nil)	`(C D E G A)`
(nil d nil nil nil)	`(C D E G A)`
(nil e nil nil nil)	`(C D E G A)`
(nil g nil nil nil)	`(C D E G A)`
(nil a nil nil nil)	`(C D E A)`

Table 2. Outputs for various starting notes of a net trained with a longer melody.

I tried varying the number of epochs the network was trained with, as well as the learning rate, forgetting rate, and markov order. None of these changes significantly improved the output. I also tried adding hidden layers to the net and initializing it randomly, and there was still no improvement in the results.

Moving on to chant melodies

As a source for Gregorian chant music, I used a reprint of an old liturgical book called the Antiphonale Monasticum. It is a collection of all of the music which has traditionally been used as a part of daily prayer by the Catholic Church. In particular, I chose three melodies which are similar to each other because of the scale which they use (mode 1 – the “Dorian” mode) and because of their purpose within the liturgy as antiphons before the singing of the Magnificat prayer during Vespers (evening prayer). The melodies I chose are from the liturgies for Tuesday of Holy Week and first vespers of the first and third Sundays of Advent (Antiphonale Monasticum 394, 186, 203).

The melodies were encoded for the Dorian scale representation of code listing 5 and the corresponding note lists are shown in code listing 6. The representation of the melodies includes the various pauses and measure lines that occur in chant music as the symbols QBAR, HBAR, BAR, and DBAR. It was hoped by including these musical gestures in addition to the note pitches that the network would learn the characteristic patterns that occur at the beginning and ends of phrases in chant.

I trained a new neural net with these melodies using a markov order of six and ten epochs. I then generated several melodies, two of which are shown in table 3. None of the melodies created really reflected the style of the chant input melodies. They frequently contain large leaps between pitches which is not characteristic of chant and the pause symbols occurred much more frequently than normal and at ackward points in the melodies. Once again, various parameters for initializing the network and various net architectures were tried in an attempt to

improve the results. Varying the learning and forgetting rates did have some effect on the training. But when I examined the outputs of the net for individual inputs, I generally found that it was an "all or nothing" situation. For some learning and forgetting rates, the network would "learn" all output notes as possibilities for most input combinations and for others it would learn none of the output notes. So it was a choice between nearly random output or no output at all. I attempted at one point to "optimize" the forgetting rate by writing a function which iteratively trained new networks while narrowing the lower and upper bounds on the forgetting rate, searching for a value which would give something in between "all or nothing" as output. A value was never found within six or seven digits of precision for the forgetting rate.

Input	Output
`(generate-melody *dorian-mode* 'd4 cn 62)`	`(D4 G4 D4 HBAR C4 G4 BF4 F4 G4 B4 D4 D4 D4 C4 A4 E4 QBAR HBAR BF4 F4 C5 F4 BAR B4 G4 D5 D5 BF4 HBAR BF4 HBAR A4 F4 BF4 C4 D5 C4 G4 D5 F4 D5 F4 C4 BF4 QBAR BAR F4 C4 A4 D5 HBAR D4 C5 BAR G4 B4 C4 G4 E4 D5 B4 A4 B4 E4)`
`(generate-melody *dorian-mode* 'c4 cn 40)`	`(C4 E4 A4 G4 G4 B4 C5 B4 C5 D5 A4 G4 E4 D5 D5 A4 C5 C4 A4 C5 D4 A4 BF4 G4 C5 BF4 BF4 E4 B4 D4 C5 QBAR BF4 QBAR C5 F4 B4 C4 HBAR BF4 HBAR F4)`

Table 3. Two melodies generated by a network trained with chant.

Wrapping it up

The output of the neural networks which I trained to replicate Gregorian chant melodies is very poor. It does not contain any of the signature patterns contained within the training melodies and does not even imitate the simple stepwise motion of most chant. Instead it leaps about rather randomly and stutters due to the frequent and unnatural occurrences of the "pause gestures." While the results of this project were less than successful, I do not believe that it means that the premise of the work is untenable. Other researchers have had more success with training neural networks to compose music that stylistically resembles its input than I have had here. For example, Mozer has had some success with using a "recurrent autopredictive connectionist network" to compose new melodies in the style of J.S. Bach and European folk melodies (Mozer 1). Chen and Miikkulainen have used a simple recurrent network architecture to imitate the work of Bartok (Chen and Miikkulainen 2). And others have been able to imitate the harmonic style of other bodies of music using neural networks.

I think that my work should be continued and more experimentation is needed. Other network architectures may prove to be better suited than the one I used here. While the feedback mechanism I used allowed for some context saving, an architecture like the simple recurrent network which has an special internal and hidden layer of nodes which feedback to the normal hidden layer may provide better results. I also think that it would be better to use far

more melodies for training the network. Around the order of 50-60 would likely be better than the three I tried in this project.

References

Catholic Church. Antiphonale Monasticum Pro Diurnis Horis. Tournai: Desclée, 1934.

Chen and Miikkulainen. “Creating Melodies with Evolving Recurrent Neural Networks.” In *Proceedings of the 2001 International Joint Conference on Neural Networks.* Washington, DC: IEEE, 2001.

Cope, David. Computers and Musical Style. The Computer music and digital audio series, V.6. Madison: A-R Editions, 1991.

Graham, Paul. ANSI Common Lisp. Prentice Hall Series in Artificial Intelligence. Upper Saddle River: Prentice Hall, 1996.

Hiley, David. Western Plainchant: A Handbook. Oxford: Clarendon Press, 1993.

Negnevitsky, Michael. Artificial Intelligence: A Guide to Intelligent Systems. Harlow: Addison-Wesley, 2002.

Mozer, Michael C. “Neural network music composition by prediction: Exploring the benefits of psychoacoustic constraints and multiscale processing.” *Connection Science*, 1994.