Music Composition Using Recurrent Neural Networks and Evolutionary Algorithms

Calvin Pelletier
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign

May 2017

Abstract

The ability to generate original music is a significant milestone for the application of artificial intelligence in creative fields. In this paper, I explore two techniques for autonomous music composition: recurrent neural networks and evolutionary algorithms. Both methods utilize data in the Nottingham Music Database of folk songs in ABC music notation. The neural network inputted and outputted individual ASCII characters and quickly learned to generate valid ABC notation from training on the dataset. The fitness function for the evolutionary algorithm was evaluated by comparing various characteristics of the generated song to target characteristics calculated using the dataset. Both methods are capable of composing homophonic music consisting of a monophonic melody and a chord progression.

Subject Keywords: artificial intelligence; music composition; machine learning; artificial neural network; evolutionary algorithm

Contents

1. Neural Network Background

Artificial neural networks (ANNs) have been used extensively in complex optimization problems involving supervised learning. ANNs are loosely analogous to the operation of biological neurons from which they get their name. Individual cells are organized into layers where the outputs of the previous layer are the inputs of the next layer, as illustrated in Figure 1.

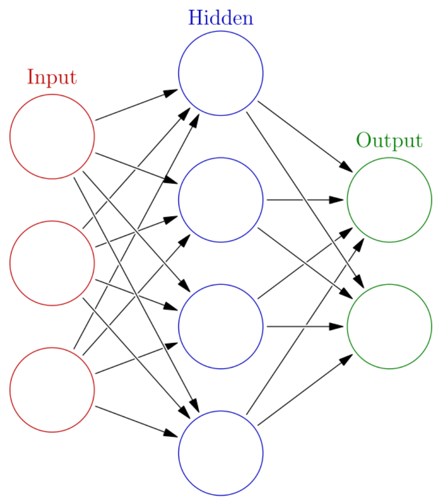

Figure 1. Layered structure of an artificial neural network.

Recurrent neural networks (RNNs) are useful when past information is required beyond the current set of inputs into the network. This matches well with music composition since there are numerous other factors involved beyond what was played in the previous time step. A basic RNN cell uses the following equation to calculate its output:

$$f(x) = K(b + \sum_{x_i \in x} w_i x_i) \tag{1.1}$$

where *x* is the set of inputs to the cell, *b* is the bias of the cell, *w* is the weight associated with a specific input, and *K* is an activation function. Inputs and outputs of the cells are between 0 and 1 or -1 and 1, depending on which activation function is used. The two most common activation functions are the hyperbolic tangent function and the sigmoid function. The sigmoid function is defined as follows:

$$S(x) = \frac{1}{1 + e^{-x}} \tag{1.2}$$

What differentiates a basic RNN cell from an ANN cell is that a basic RNN cell's input includes the output of that cell from the last time step, thus allowing the network to retain information. See Figure 2 for an illustration.

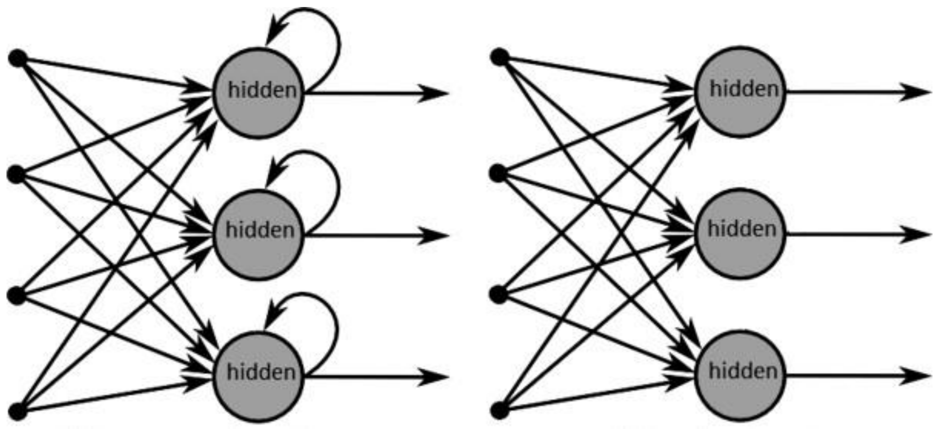

Figure 2. Comparison of a RNN (left) and an ANN (right).

The network predicts an output by either selecting the output cell with the maximum output value or by selecting one randomly with the probability distribution given by the softmax of the output values. The latter is more applicable to music composition as a way to promote variety in the song. The softmax function is the log normalizer of a categorical probability distribution which converts a *K*-dimensional vector *x* of arbitrary real numbers to a *K*-dimensional vector of real numbers between 0 and 1, exclusive, that add up to 1.

$$softmax(x) = (\sigma(x)_i, \ldots, \sigma(x)_K), \qquad \sigma(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}} \tag{1.3}$$

During training, the neural network attempts to minimize the cross entropy of the predicted distribution, y, and the actual distribution, *y'*. The cross entropy equation is

$$H_{y'}(y) = -\sum_i y'_i \log(y_i) \tag{1.4}$$

To minimize the cross entropy, a gradient descent algorithm is used to update the parameters. The trainable parameters in a basic neural network are all the weights and biases. The simplest optimization algorithm is stochastic gradient descent, which determines the new value of a parameter p from its old value, a learning rate ε, and the gradient of the cross entropy with respect to the parameter.

$$p := p - \varepsilon \nabla H(p) \tag{1.5}$$

2. Composition Using RNNs

2.1 MIDI Training Data

My first attempt at composing music with an RNN involved generating music from a dataset of MIDI songs. I converted the MIDI songs into training data by replacing the event-based structure with a timeline-based one. The timeline consisted of time steps, each with a quarter of a beat duration (i.e. 16 time steps per measure for a song in 4/4 time). The input included the beat (4 bits representing the location in the measure) and the notes that were played in the previous time step. To differentiate between the same note being played multiple times in a row and a note being held for multiple time steps, a time step also includes information about whether the current note is an extension or a re-articulation of the previous note. Since each note needs 2 bits and there are 128 possible MIDI notes, the input into the neural network was 260 bits wide. The output was 256 bits wide since the beat information was calculated outside the neural network. Since multiple notes could be played at once, there were 2^{256} possible outputs for each time step. Unsurprisingly, this method was unsuccessful in creating decent-sounding music. The songs composed by this neural network were overwhelmingly dissonant and lacked a clear structure.

2.2 ABC Training Data

To reduce the number of outputs, I transitioned to generating homophonic music (single melody with a chord progression) from a dataset of songs in ABC notation. Since ABC is entirely text based, this method utilized a character based RNN by inputting one character each time step and predicting the next character. The only preprocessing done was using TensorFlow's built-in support for character embedding to limit the neural network's vocabulary to just the characters seen in the training data. During composition, the RNN is primed with "X:", which is how ABC notation always starts, then it sequentially predicts characters until two newline characters are predicted in a row, which is how songs were separated in the dataset. Following is an example of a song in ABC notation:

```
X: 330
T:Irish Whiskey
S:Trad, arr Phil Rowe
M:6/8
K:G
B|"G"G3 "C"g2e|"G"dBG "D"AFD|"G"G3 "C"g2e|"G"dBG "D"A2B|
"G"G3 "C"g2e|"G"dBG GAB|"C"cde "G"dcB|"D7"cBA "G"G2::
B|"Em"eBe gbg|eBe gbg|"Em"eBe "Bm"g2a|"Em"bag "D"agf|
eBe gbg|"Em"eBe "Bm"g2a|"Em"bag "D"agf| [1"Em"e3 -e2:|[2"Em"e3 "D7"d2||
```

Notes are selected with the letters 'a' through 'g' and preceded by special characters for accidentals. The octave is selected with the case of the letter and with trailing commas/apostrophes. The default duration of a note is specified in the meter and altered by the numbers after the notes.

2.3 Hyperparameter Optimization

I separated the dataset into 90% training data and 10% validation data. I chose not to also separate it into testing data because, in this specific application, the testing error provides no insight into the quality of music that the RNN produces. I used the RNN's performance on the validation data to hand-optimize various hyperparameters in the model, including number of layers, cells per layer, learning rate, optimization algorithm, exponential decay rate (for Adam optimization), and RNN cell type.

For the optimization algorithm, I explored the effectiveness of stochastic gradient descent, Adadelta (Zeiler, 2012), and Adam (Kingma and Ba, 2014). I found Adam, with the learning rate at 0.002 and the decay rate at 0.97, to be the most effective at minimizing error. Adaptive Moment Estimation (Adam) stores exponentially decaying averages of past gradients, m, and of past squared gradients, v, using the decay rates β_1 and β_2. The equations for updating the averages and parameters are

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla H, \qquad v_t = \beta_2 v_{t-1} + (1-\beta_2)\nabla H^2 \tag{2.1}$$

$$p := p - \frac{\eta}{\sqrt{\frac{v}{1-\beta_2}} + \varepsilon} * \frac{m}{1-\beta_1} \tag{2.2}$$

For the RNN cell type, I found Long Short-Term Memory (LSTM) (Hochreiter and Schmidhuber, 1997) cells to be the most effective among LSTM, GRU, and basic RNN cells. Traditional recurrent neural networks struggle with handling long-term dependencies. Remembering information from a large number of time steps ago is especially important in this application because ABC notation establishes crucial information in the header, such as the key, which needs to be remembered throughout the entire song. LSTM RNNs are well suited for this constraint because the LSTM cell introduces the capability of learning long-term dependencies. Figure 3 is a time-unraveled LSTM layer featuring sigmoid and hyperbolic tangent ANN layers and point-wise arithmetic.

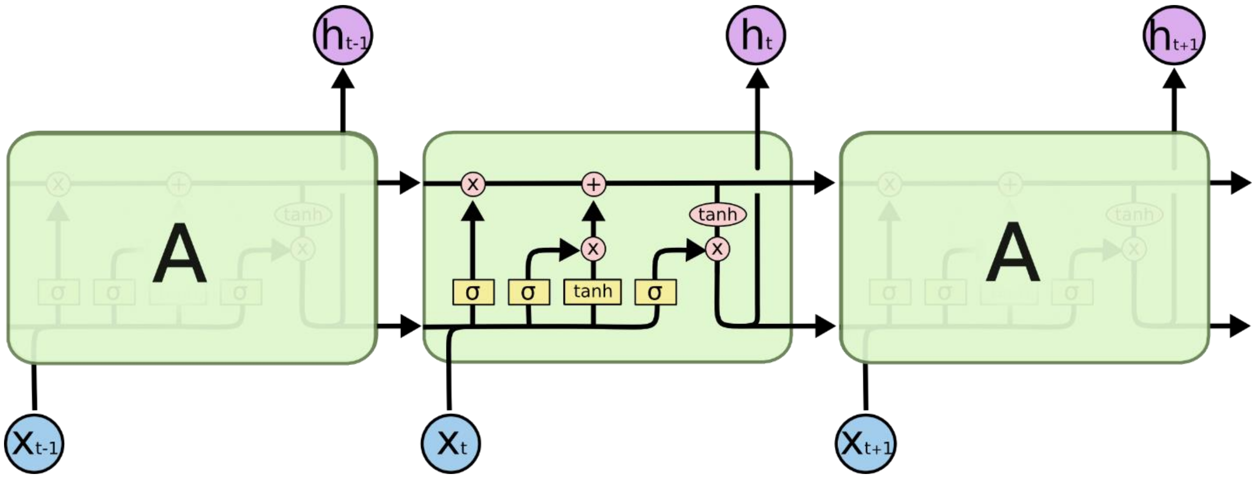

Figure 3. Structure of a time-unraveled LSTM layer.

The key to LSTMs is that the passage of the cell state through time is regulated by structures called gates, thus giving the network more control over what it remembers and forgets. Gates consist of a sigmoid neural network layer and a pointwise multiplication operation. There are three gates used in standard LSTMs: a forget gate which decides what information to discard from the cell state, an input gate which decides what to update in the cell state, and an output gate which decides what parts of the cell state to output. There are many variants of LSTM, most notably the Gated Recurrent Unit (GRU) (Chung, Gulcehre, Cho, and Bengio, 2014) which simplifies LSTM by combining the forget and input gates and merging the hidden state with the cell state.

Optimizing the network size (number of layers and cells per layer) consisted of finding a balance between enough complexity to classify the data and too much complexity which would result in overfitting. Overfitting can be identified when the validation error starts to retrogress despite the training error continuing to improve. The graph in Figure 4 compares the validation error over time for the optimal number of cells per layer, 138, to other values for this hyperparameter.

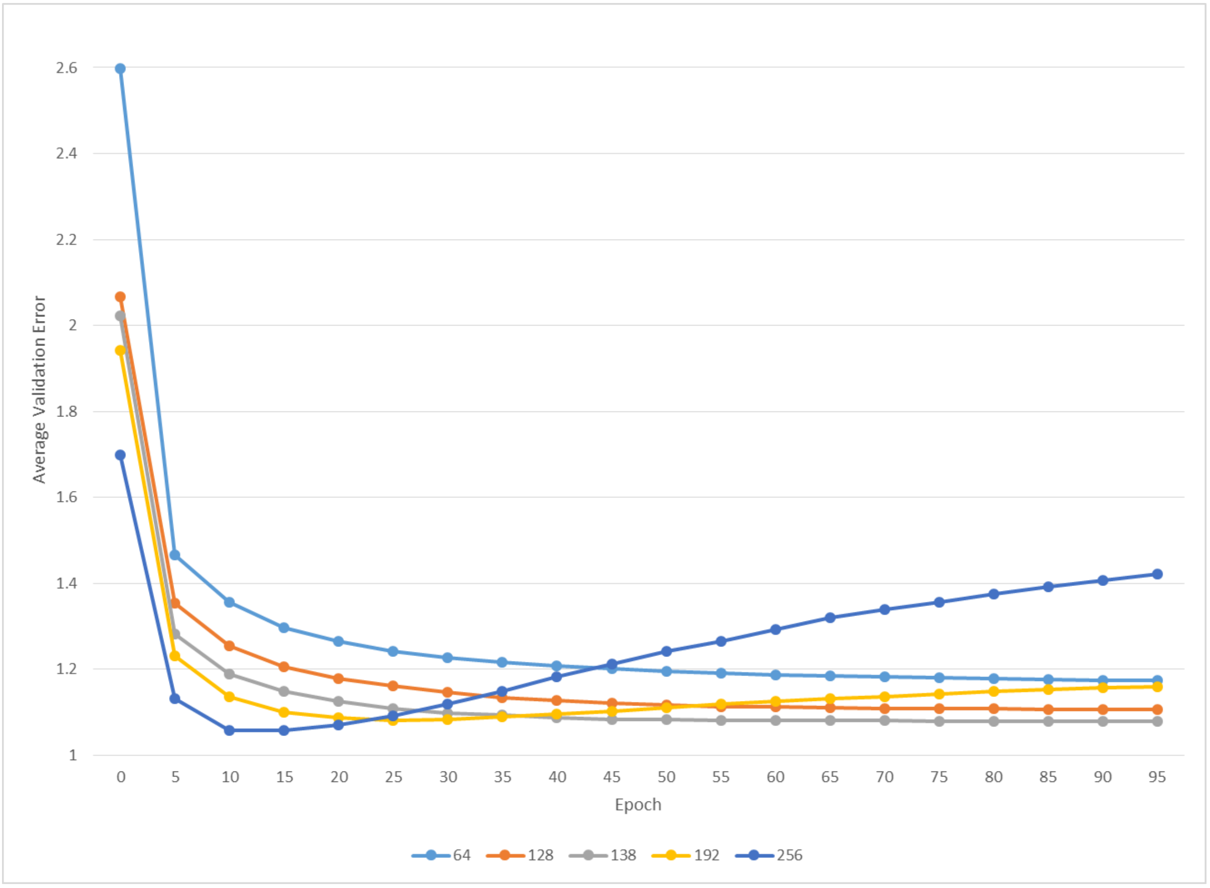

Figure 4. Performance on the validation data of networks with varying cells per layer.

I achieved the lowest validation error with 2 layers of 138 LSTM cells each, optimized with Adam at a learning rate of 0.002 and a decay rate of 0.97. The average training and validation cross-entropy during each epoch for this model is given in Figure 5.

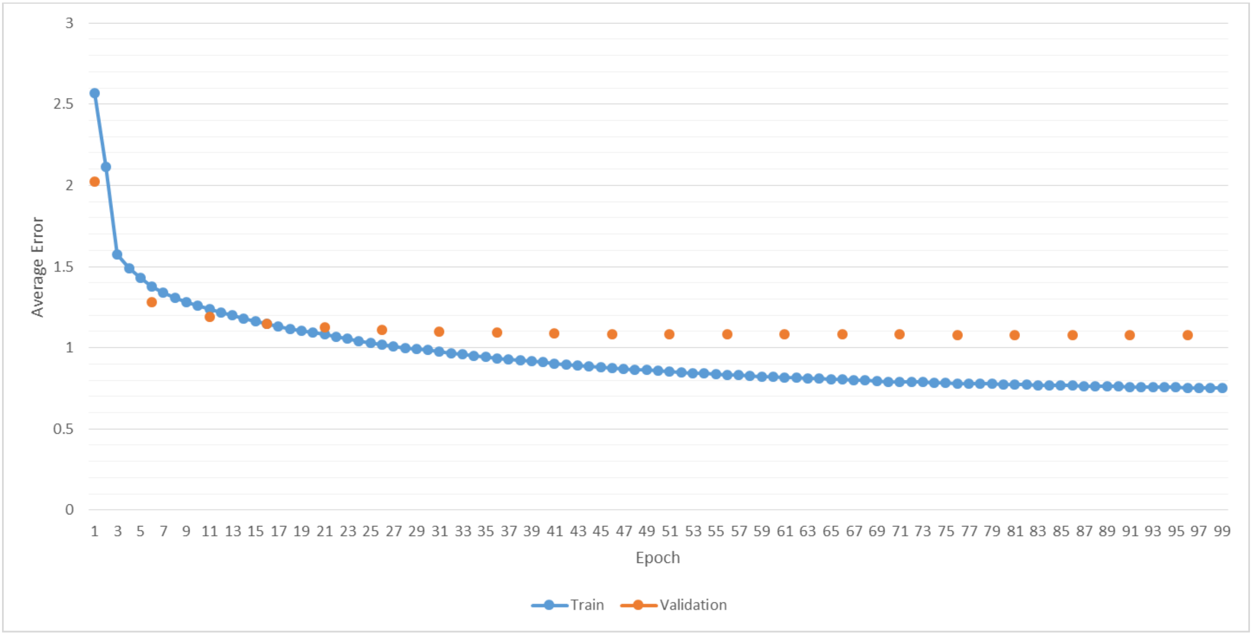

Figure 5. Performance of the optimal network on the training and validation data.

2.4 Composition

The following compositions are sampled from the network shown in Figure 5 after 1, 10, and 100 epochs, respectively.

```
X: ^-"D"gb|"G"d32|"D"GB|de "G"e2de deA3pE=e2F "F"B2B A/2c"A/2A/2e/4|\"G"GB
"gm"AB|"A"cA 5G:B2 "D"B3a "D"d2G|"|"D"dEF G"c|Ac G
"D"AD"g,dFc acB/2|"D"f2c2_d/2|"A7BA|"A"f3_ "B"f2BA|
"Em"AA"d/:
F8"A"fG3 \"G"g6fd|"e7"ed d:
""D:afa "D"ena eE|

X: 2
P:AyB/4c/4"G"d f/2d/2|"G"Bd "A"eA|"G"d/2d/2e/2f/2 "G"g3/2f/2|"G"g2 ed|"D7"AB/2c/2
dc/2B/2|\
"F"F/2c3/2 A3/2D/2|"G"BB Bc/2B/2|"D"d3/2f/2 fa|"G"g3/2e/2 ^f/2e/2c/2c/2|\
"A"cA A2|"D"d2 dA/2B/2|"G"BG G/2F/2c/2d/2|
"G"B/2B/2A "G7"A/2B/2-|"G"GB d/2e/2g/2f/2|ed e/2f/2e/2d/2|"A"e2 A2|"G"de
"D7"f/2a/2d'/2f/2|\
"E7"=f2 ed/2G/2|"A"A/2A/2g/2d/2 cd|\
"D"dA "B7"GG|
"D"A/2d/2c/2d/2 fe/2d/2|"G"Bd|"G"B/2B/2A/2B/2 d/2B/2A/2c/2|"Em"BB/2c/2 dB/2c/2|\
"G"B/2^G/2A/2B/2 "G"Bd/2c/2|"G"B/2A/2B/2B/2 B/2c/2d/2d/2|"G"dd/2

X: 10
M:6/8
K:D
"A"c2E "D"FDE|"D"DFA dFA|"D"FED F2G|"A7"A2d "G"B2G|
B3 dcB|"D"AGF F2A|"C"ed4|"G"B3 d3|"C"e2e "G"d2d|"Em"e2f "Em"e2d|
"A7"c2e cBA|"Bm"B2^c "a"d2B|"Em"B2A B"A"AB|"D"A2A A2d|"Em"B2G G2B|"D"dAF A3|"D"d3 f2g|
"D"f2e d2f|"Em"g2d e2d|"A7"edc "D7"d3|"G"d2d Bcd|
"Am"c2e a2"A7"e|"D7"f2f A2e|"D7"d2A FAc|"D7"d4-A2|
"D7"def edc|"G"d2G GB"D7"A|"G"dBG "D7"g2=c|"G"B2G G2:|:B/2g/2||
```

The composition from epoch 1 is entirely invalid, though it resembles ABC notation. At epoch

10, the neural network produces mostly valid notation, but there are a few scattered errors and it leaves out essential information from the header. By epoch 100, the majority of compositions are valid.

3. Composition Using an Evolutionary Algorithm

3.1 Overview

Evolutionary algorithms are metaheuristic optimization algorithms. They are utilized in a variety of applications from the vehicle routing problem (Prins, 2004) to process scheduling in manufacturing systems (Kim, Park, and Ko, 2003). They work by mutating individuals in a population, then selecting the best according to a fitness function and repeating. In this case, songs were individuals and the fitness function was an attempt to algorithmically determine the quality of a song. Generally, evolutionary algorithms are limited by computational complexity, since a massive number of individuals need to be evaluated by the fitness function. However, in this application, the algorithm is limited by the ability of the fitness function to differentiate between a good and bad song.

In contrast to the black-box nature of artificial neural networks, this technique allowed me to be more creative in designing the algorithm. Rather than rely on a neural network to find common patterns in music, I designed the fitness function to capture what I thought were the most important features of a song.

I used a different format to structure songs for the evolutionary algorithm than I did with the RNN in order to limit the search space and reduce the time needed to evaluate the fitness function. Songs were divided into time steps, each the length of a quarter of a beat. During each time step, there is one chord and either a note or an extension of the previous note. The chord progression, time signature, song length, and key are all static characteristics of the song and are chosen randomly. The melody is initialized randomly and is subject to mutation.

3.2 Fitness Function

The fitness function is evaluated by comparing various characteristics of the song to target values. These characteristics are energetic, progression dissonant, key dissonant, rhythmic, rhythmically thematic, tonally thematic, range, and center. These characteristics were chosen to capture the basic rules of music theory. I defined the energy of a song as follows:

$$\frac{\sum_{i=1}^{N-1} \frac{dist(x_i, x_{i+1})}{d_i}}{N} \tag{3.1}$$

where *N* is the number of notes in the melody, *dist* is the distance between two notes in semitones, and d_i is the duration of a note in time steps. The equation below is used in calculating progression dissonance, key dissonance, and rhythm:

$$\frac{\sum_{i=1}^{N} Q(x_i) * d_i}{L} \tag{3.2}$$

where *L* is the length of the song in time steps and Q is specific to the characteristic. For progression dissonance, Q evaluates to 0 when a given note is within the chord being played during the same time and 1 otherwise. For key dissonance, Q evaluates to 0 when a given note is within the key and 1 otherwise. For example, if the song is in the key of A and a C sharp major is being played, the notes A, B, C sharp, D, E, F sharp, and G sharp are in the key and the notes C sharp, F natural, and G sharp are in the chord. For calculating how rhythmic a song is, Q depends on the time signature and on which beat within the measure the note starts. Essentially, down beats evaluate to 1, off beats to 0.5, and further subdivisions to 0.

The rhythmically and tonally thematic characteristics analyze the amount of repetition in the melody. The pseudo-code for these functions is shown below. The "press" attribute of a time step indicates whether the current melody note was played directly during that step or was an extension of a previous step. The "stringify" function converts a sequence of notes into a string for comparison purposes. The resulting string contains information about the pitches and order of the notes but not the durations. N is the number of steps, L is the number of notes in the melody, M is the number of measures, and K is the number of steps per measure.

```
RhythmicallyThematic($steps[0..N-1]$):
  $measures[0..M-1][0..K-1]$
  for $i \leftarrow 0$ to $N-1$:
      if $steps[i].press = \text{TRUE}$:
          $measures[i/K][i \bmod K] \leftarrow 1$
      else:
          $measures[i/K][i \bmod K] \leftarrow 0$
  $total \leftarrow 0$
  for $i \leftarrow 0$ to $M-2$:
      for $j \leftarrow i$ to $M-1$:
          $diff \leftarrow 0$
          for $k \leftarrow 0$ to $K-1$:
              if $measures[i][k] \neq measures[j][k]$:
                  $diff \leftarrow diff + 1$
          if $diff = 0$:
              $total \leftarrow total + 1$
          else if $diff = 1$:
              $total \leftarrow total + 0.3$
          else if $diff = 2$:
              $total \leftarrow total + 0.1$
  return $total/M$

TonallyThematic($notes[0..L-1]$):
  $total \leftarrow 0$
  $sequences \leftarrow \emptyset$
  for $i \leftarrow 0$ to $L-2$:
      for $j \leftarrow i+1$ to $\text{MIN}(L-1, i+1+K)$:
          $s \leftarrow \text{STRINGIFY}(notes[i..j])$
          if $s \in sequences.keys$:
              $sequences[s] \leftarrow sequences[s] + 1$
          else:
              $sequences[s] \leftarrow 1$
  for all $(k, v) \in sequences$:
      $total \leftarrow total + v$
  return $total/L$
```

Finally, range and center are simply the range of notes used in the melody (in semitones) and the average MIDI value (middle C is 60) of the notes in the melody weighted by duration, respectively.

To calculate the target values, I converted the dataset used in training the RNN to the format used in the evolutionary algorithm and analyzed the distribution of values for each characteristic. See Table 1.

Table 1. Distribution of values for various characteristics of the songs in the ABC dataset.

Characteristic	Average	Std. Dev.	Minimum	Maximum
Energetic	0.952	0.314	0.214	2.66
Progression Dissonant	0.232	0.0669	0.0147	0.559
Key Dissonant	0.0124	0.0248	0.0	0.195
Rhythmic	0.843	0.0992	0.623	1.0
Rhythmically Thematic	7.22	13.3	0.963	279
Tonally Thematic	9.71	3.74	0.415	15.9
Range	16.8	3.27	7.0	27.0
Center	73.4	8.87	59.0	83.2

The target values were actually target ranges within one standard deviation of the average. The final output of the fitness function was the sum of the distances from the target range for each characteristic. Each distance was weighted by the inverse of that characteristic's standard deviation so that they equally contributed to the fitness function.

3.3 Composition

During composition, the songs in the population were mutated and selected based on the lowest fitness function score, iteratively, until a song was entirely within the target ranges. Mutation consisted of adding/removing notes, transposing notes, and increasing/decreasing the duration of notes. This process generally took around a few thousand generations at a population size of 100. Figure 6 is a song generated using this technique.

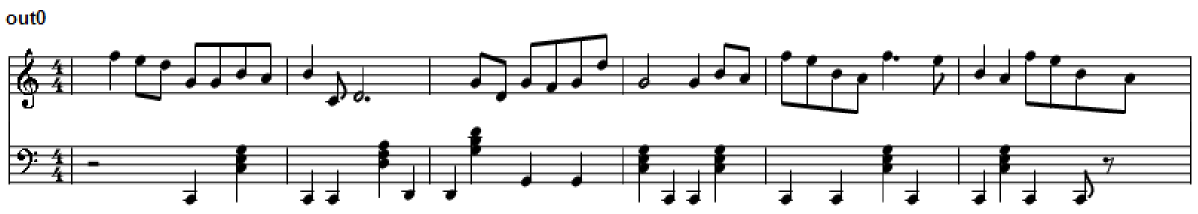

Figure 6. A sample output from the evolutionary algorithm.

4. Results

To compare the effectiveness of these music composition techniques, I asked 15 people to rate songs on a scale from 1-5 and specify whether they thought it was composed by a computer or human. 1/6 of the songs were actual folk songs from the dataset used throughout this paper. Another 1/6 were written by the evolutionary algorithm and converted to WAV. The remaining 4/6 were written by four different RNNs, each trained on a different manipulation of the dataset, and converted from ABC to WAV.

The four different versions of the dataset were the original, regularized, reversed, and reversed-regularized. For the regularized, I transposed every song in the dataset into the key of C. The reasoning for this is that the neural network would no longer need to maintain knowledge of the key throughout composition. Normalizing the key would not limit the neural network's ability to write a wide range of music since most composers agree that the exact key has very little importance. The reasoning behind reversing the training data is to explore the effectiveness of composing past notes from information about future notes rather than the other way around. The reversed-regularized version combines both of those manipulations. Each version uses a slightly different RNN model because the hyperparameters are optimized separately for each one. Table 2 gives the results (sample size is 450). "% human/computer" are the results to the question, "If you had to guess, do you think a computer or human wrote this song?".

Table 2. Results of the survey.

Technique	**Avg. rating**	**Std. dev. of rating**	**% human/computer**
Human-composed	3.91	0.906	60.5/39.5
Evolutionary algorithm	2.87	0.896	25.4/74.6
RNN (original)	3.46	1.09	44.4/55.6
RNN (regularized)	3.13	0.884	29.3/70.7
RNN (reversed)	3.13	1.12	40.2/59.8
RNN (reversed-regularized)	3.54	0.779	33.3/66.7

As expected, actual songs were rated highest. Interestingly, the regularized and reversed RNNs underperformed the original but the reversed-regularized RNN outperformed it. Both the regularized RNNs had a significantly lower standard deviation than the non-regularized RNNs, likely due to the reduced variability in the training data.

5. Conclusion

While the algorithmically composed songs did underperform the human composed ones, these techniques could be quite effective as a compositional aid to musicians. Furthermore, there are many potential improvements to these methods. For example, dropout could be applied to the RNNs to prevent overfitting, thus allowing additional complexity. For the evolutionary algorithm, many of the details were chosen arbitrarily. I plan on tweaking the specifics of the algorithm and exploring additional characteristics to add to the fitness function.

References

Chung J, Gulcehre C, Cho K, Bengio Y. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence. CoRR. 2014; 1412(3555).

Hochreiter S, Schmidhuber J. Long Short-Term Memory. Neural Comput. 1997; 9(8):1735–80.

Kim Y, Park K, Ko J. A Symbiotic Evolutionary Algorithm for the Integration of Process Planning and Job Shop Scheduling. Computers & Operations Research. 2003; 30(8):1151-71.

Kingma D, Ba J. Adam: A Method for Stochastic Optimization. CoRR. 2014; 1412(6980).

Prins C. A Simple and Effective Evolutionary Algorithm for the Vehicle Routing Problem. Computers & Operations Research. 2004; 31(12):1985-2002.

Zeiler M. ADADELTA: An Adaptive Learning Rate Method. CoRR. 2012; 1212(5701).