Using Recurrent Neural Networks to Judge Fitness in Musical Genetic Algorithms

Peter Mitrano, Arthur Lockman, James Honicker, Scott Barton
Worcester Polytechnic Institute
mitranopeter@gmail.com, hello@rthr.me, jlhonicker@wpi.edu, sdbarton@wpi.edu

Abstract

We used a recurrent neural network as a fitness function for a genetic algorithm to generate monophonic solos. The genetic algorithm is based on GenJam as described in Biles (1994). We conducted training sessions with human participants in order to compare and quantify some of the differences between human-feedback and RNN fitness functions. We found that the RNNs can effectively play the role of human fitness feedback, but still suffer in many areas. Our results suggest that certain types of recurrent neural networks can address the issues with human feedback, and thus should be explored in future research.

There have been many approaches to automatic composition that combine AI techniques. Often these systems contain a generative component and an optimizing component. Techniques for the generative component include genetic algorithms (Biles, 1994; Loughran & O'Neill, 2016), feed-forward neural networks (Bickerman, Bosley, Swire, & Keller, 2010), recurrent neural networks ("Google Magenta," n.d.)), and grammars (Gillick, Tang, & Keller, 2010; Keller & Morrison, 2007). The optimizing component could be gradient descent for neural network generators (Geoffrey E. Hinton, D. E. Rumlhart, & Ronald J. Williams, 1988), more genetic algorithms (Loughran & O'Neill, 2016), human input (Biles, 1994), or variable neighborhood search (Herremans & Chew, 2016).

Genetic Algorithms

Of the aforementioned approaches, genetic algorithms (GA) have shown promising results and thus have been used in a number of automatic composition systems over the past 25 years (Gibson & Byrne, 1991). Genetic algorithms consist of a population of solutions to a problem and a fitness function that is used to rank those solutions. Solutions are iteratively ranked, crossed with each other to produce a new generation, and then mutated. As this process is repeated, the fitness of the solutions in the population increases. An important aspect of designing a genetic algorithm is choosing a representation for the *genotype* and *phenotype*. Genotype refers to how the data is encoded in the computation space so that it can be manipulated in mutation and crossover, and is usually a simple list of numbers. The phenotype is the expression or decoding of that data in a form that is relevant to the solution-finding process, and in this case is the music that the fitness function receives. Choosing how the genotype is decoded to the phenotype affects what musical factors are fixed and what factors are learned, and it also affects the size of the solution space that is being optimized. Part of the appeal of genetic algorithms for generating music is that they can have a simple genotype that maps to the output music, rather than directly representing all the parametric complexities that the music may contain. Additionally, genetic algorithms can be endowed with domain specific information about musically relevant attributes and relationships in the form of mutation, crossover, and selection methods. Further, they allow the amount of randomness in the output to be adjusted in the form a mutation rate. In a musical genetic algorithm, these mutations often result in meaningful manipulations, such as repeating a section, reorganizing a chord, or reversing a sequence of notes.

Musical Fitness Functions

The fitness function is an important part of a musical genetic algorithm, as it determines what solutions are deemed "good". The problem is that in music, "good" solutions are subjective and highly dependent on context. The extent to which a particular sequence of pitches will work is not inherent either in the pitches themselves or their organization; it is also a function of harmonic context, rhythmic configu-

ration, the pitches that precede and follow it, expressive timing, dynamic contour, and timbre. Further complicating the issue is that perception of musical organizations is affected by interactions between musical parameters. In some cases, the rhythm of a musical gesture will dictate perceptual grouping and in others it may be the pitch or timbral sequence. As humans, such perceptual organization is primary to our aesthetic evaluation of musical gestures. One must also consider how to allow the output to feel creative, as opposed to a satisfactory regurgitation of rule-based conventions. Even when a good fitness function is written, it likely cannot be reused if the kind of desired musical output changes. For instance, a fitness function that knows the rules of baroque four-part harmonic voice leading would likely have little overlap with a fitness function that judges the quality of a Dubstep Wobble Bass. Designing a function that captures the ambiguities and complexities of such processes and requirements would be exorbitantly difficult, if possible at all.

One way to address the aforementioned issue is to use human feedback as the fitness function. In these *interactive genetic algorithms* (IGA), a human evaluates each chromosome that is produced. Interactive genetic algorithms excel because they incorporate human aesthetic preferences, which influence the output of the system in the direction of more pleasing results. The drawback of this process is the tedious effort on the part of the human trainer. An additional concern is the consistency of human feedback, which may vary because of a number of factors including attention, repetition, sequence effect, context, and fatigue. The ways in which the consistency of human feedback affects fitness in interactive genetic algorithms needs to be further clarified.

To avoid the tedious and inconsistent results of a human-feedback fitness function, one could derive an automatic fitness function by learning from existing music. The hope is that a learned function will still follow conventions, but without being restricted to the rules a programmer can create. One such data-driven fitness function is a *recurrent neural network* (RNN). In a *feedforward neural network*, the output is a function of only the most recent input. If one note was passed in, the next note would be predicted based only on that note, similar to a 1st order markov chain. In an RNN, the output is function of all previous inputs: an infinite order markov chain. One challenge of automatic fitness functions for music is how to encode the input in a machine readable format. For RNNs, the input is typically a representation of a sequence of pitches and durations. In the case of a MIDI representation, the sequence is encoded as note on, note off and velocity events that are fed into the RNN. Each event could also be accompanied by human-labeled information, such as chord progression, which is needed to predict the next event. In most cases, RNNs have been used in music to generate music, rather than to judge it (Chen & Miikkulainen, 2001; "Google Magenta," n.d.). Specifically, musical RNNs often end with a softmax layer with one class for each possible note. Softmax is a function that converts the output of the previous layers into a probability distribution over the possible notes. The node in the output of the softmax layer with the highest activation represents the next predicted note. In this way, a musical sequence can be produced given a starting note or set of starting notes. This architecture makes it straightforward to generate monophonic melodies.

Neural networks have been combined with genetic algorithms in a number of ways. Biles, Peter, and Loggi (1996) used a small feed forward neural network to learn the fitness of single measures and phrases given only the genotype (indexes), not the phenotype (pitches). Genetic algorithms have been paired with RNNs to maximize a rule-based fitness function (Chen & Miikkulainen, 2001). In this approach the RNN is used as generator of music, and the genetic algorithm evolves the best RNN according to a handcrafted fitness function. RNNs have also been used as a fitness function where fitness ratings were based on similarity to two melodies. In this case, the RNN acted as a distance function between the generated music and the two example pieces (Sheikholharam & Teshnehlab, 2008). Our approach uses an RNN as a fitness function, where the RNN was trained on a large music corpus with the goal of judging musical fitness in a broader sense.

Current Study

Writing a fitness function for a genetic algorithm is difficult, and using human feedback as the fitness function is tedious and potentially inconsistent. Given the features of RNNs and their successful combination with genetic algorithms in other scenarios, we wonder if an RNN used to give fitness feedback to a genetic algorithm music generator would yield easier and more promising results. Our project thus seeks to achieve the following objectives:

1. Use an RNN to provide fitness feedback to a genetic algorithm music generator
2. Compare human-feedback fitness functions to RNN fitness functions to determine the strengths and weaknesses of each

Design

Genetic Algorithm for Music Generation

The genetic algorithm in our system is based on the description of GenJam provided in Biles (1994), but has a number of key differences. We used population sizes on the order of 256 measures and 256 phrases, as opposed to 32 and 48 as

used in GenJam. Larger populations create more opportunities for a high-fitness gene to be found. We also included an additional random-bit mutation operation, which unlike the other mutations does not attempt to preserve musical qualities or manipulate them according to conventions of harmony and rhythm. This can be tolerated by the RNN, but was avoided in human training to avoid subjecting the listener to the results of mutations that are unlikely to improve the gene. Another difference in our system is that fitness is assigned to measures and phrases at each generation, not accumulated over generations. We accumulated human feedback over each generation in order to smooth variable results. We assigned fitness to the log-likelihood output of the RNN. These changes were made possible because we aren't limited by the patience or concentration of human evaluators.

The parameters in our system include smallest rhythmic interval, allowed pitches, measure population size, phrase population size, number of measures per phrase, and initial note distribution. Velocity was set to a constant value during both manual and RNN training as its inclusion presents complexities in both training conditions.

Populations and Mutations

Our genetic algorithm is composed of two populations: the *phrase* population and the *measure* population. The rationale behind this is to make it easier for patterns and repetition to arise at various levels within a grouping hierarchy. Another benefit of two populations is that it permits longer sections of music to be evaluated for fitness by the RNN.

Each individual in the measure population contains pitches, rests, and sustains of the previous note or rest. Sustains are used to create notes that are held longer than the one gene. This has the advantage of keeping measure genes all the same length. Phrases are a sequence of measures, and the phrase genes are indexes to a measure in the measure population. The number of genes at the phrase level can vary according to the chord progression being used. For example, if the chord progression spans eight measures, each phrase gene would contain eight numbers that correspond to those measures. Each of these measures would, in a 4/4 time signature with one gene equaling one 8th note duration, contain eight genes. This gene duration is the smallest rhythmic interval, and is another parameter of our system (16th notes are the smallest intervals used in our experiments). Each measure-gene ranges from 0 to 15, so as to fit neatly within two bytes. 0 indicates a rest, 15 indicates a sustain of the previous rest or note, and 1-14 indicates a new note. The numbers 1-14 are an index into the list of allowed pitches for the current chord. These lists of allowed pitches control how genes in the measure genotype are mapped to actual pitches. For instance, if the current chord is a C major 7, the allowable pitches would be C D E G A B, which would create the allowed pitch map {C3 = index 1 … D5 = index 14}. Because we only allow two bytes per gene, the lists of allowed pitches for a chord can only contain 14 pitches. More bytes could be allocated and more pitches allowed, but we chose to follow the sizes used in GenJam to reduce the number of variables being manipulated. We use the same lists of allowed pitches as detailed in Biles (1994).

Like GenJam, we initialize only the phrase population uniformly randomly. For the measure population, the initialize with a fixed probability distribution of 20% rests, 20% sustains, and 60% notes, which was copied from GenJam. Unlike GenJam however, the measure genes that are notes (1-14) are not uniformly random. The first note gene in a measure is constructed by picking a number uniformly at random from 1 to 14. For all subsequent notes in that measure, a note is selected uniformly randomly from a range of plus or minus three of the previous notes. For example, if a sequence started 4, 15, 0, 15, then the next note would be randomly selected from 4 ± 3. This initialization method reduces the number of large intervals between notes. This is desirable because large intervals are less common in typical tonal melodic sequences and thus are judged with lower fitness by our RNNs, so reducing these in the initial population yields faster convergence. It also yields faster convergence for human training for the same reason. The mutation operations include sorting, shuffling, and transposing. Phrase mutations are primarily based on fitness so that either better measures or revived unused measures perpetuate. For a detailed analysis of the mutations used, our code (linked in the results section) will be the best resource, along with the descriptions provided by Biles (1994).

Recurrent Neural Networks

We experimented with three different RNNs that are available in Google's open source Magenta project ("Google Magenta," n.d.): *Improv RNN*, *Attention RNN*, and *Lookback RNN*. The *Improv RNN* model generates melodies that are conditioned on a chord progression (Simon, 2017). The other two do not incorporate accompanying chords (see Waite, 2016). All of the models used were pre-trained by Google on datasets containing thousands of songs. Training our own model on small but specific dataset, such as the Weimar Jazz DB, did not improve our results with *Attention RNN* or *Lookback RNN*. Furthermore, training *Improv RNN* requires chord labels, which were unavailable in a consistent format. We used two hidden layers with modified LSTM cells, 64 nodes per each hidden layer, and a 128-class softmax. While these networks are designed for music generation, we used them for computing fitness. To do this, we computed the log-likelihood of a given sequence generated by the genetic algorithm. In other words, we determined the forward pass of each note on/off/sustain event sequentially, and the network returned a measure of how likely it was that the event sequence came from the

distribution of its training data. This was used as the fitness value for the sequence. While the magenta team has not released the dataset used to train these networks, we know they consist of monophonic melodies with chord progressions in the case of *Improv RNN*.

Human Fitness Function

In order to evaluate the RNN as a fitness function, we compared it to one based on human-feedback. The latter is how the original GenJam was trained, and we reproduced those results here. Two of the authors participated in human training sessions. In each session, the participant listened to the output of each generation accompanied by backing chords and a monophonic bass line. The tempo of musical selections was between 100 and 160 bpm. The participant evaluated musical statements by pressing 'g' ("good") or 'b' ("bad") on a computer keyboard whenever so inspired. When 'g' was pressed, the phrase and measure that was playing three beats ago had fitness increased by 1; when 'b' is pressed, 1 was subtracted.

Because an evaluative response does not occur simultaneously with the presentation of a stimulus, a delay parameter needs to be introduced to account for the amount of time it takes a participant to register his or her feedback. We used a delay of three beats in accordance with Biles (1994), but note that the time between stimulus presentation, perception, evaluation and response is complex, variable and context-dependent. Determining an accurate measure of feedback delay was outside the scope of this particular study, and thus presents a limitation of our results.

Results

Evaluating the RNN Fitness Function

We used three inputs to evaluate the RNN as a fitness function: *random* notes, the *tonic* played repeatedly, and *professional* jazz solos. *Random* notes consisted of randomly selected pitches from the set {MIDI notes 0-127} and randomly selected rhythmic durations from the set {16th, 8th, quarter, half, whole}; all notes had a constant velocity. We used the Weimar Jazz Database as a source of professional jazz solos. We used 7 sequences of each input type with lengths between 20 and 400 notes. Only 7 sequences were tested because labeling jazz solos was time consuming. The lengths of the jazz solos were fixed, but the random and tonic sequences were given random lengths. Generating this graph multiple times over difference random samples produced consistent results. Each input sequence was run through *each RNN* and the resultant fitness was divided by sequence length. This division compensates for the fact the RNN accumulated log-likelihood with respect to length, so dividing by length makes sequences more comparable despite varying in length.

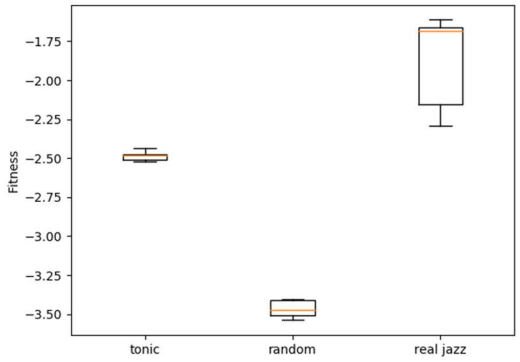

Figure 1 Improv RNN prefers professional over the tonic and random inputs

The *random* and *tonic* inputs produced lower fitness scores than the *professional* input (see Figure 1). This suggests that *Improv RNN* can indicate fitness in a basic, musically appropriate way. Conversely, both *Lookback RNN* and *Attention RNN* preferred the *random* to the *professional* input, which indicates they are not appropriate to use as fitness functions.

We also evaluated how the *Improv RNN* ranked pitches for a given harmonic context (see Figure 2 and Figure 3). We created a string of four measures of notes of various durations at the pitch we wanted to test, and labeled them as being accompanied by a C major 7 chord (C3, E3, G3, and B4). For example, the blue dot in the upper left corner of Figure 2 shows the fitness of a sequence of 16 quarter notes of the pitch C3, labeled with C Major 7 as the accompanying chord. With the current chord labeled as C Major 7, the highest fitness pitches at the quarter note duration were G followed by E and the lowest fitness pitches were C# and F#. When we compare this to models of pitch hierarchies in human perception, we see similar distinctions between components of the major triad, the other members of the diatonic scale, and the non-diatonic chromatic pitch classes (Krumhansl, 1979). These results show that although the conventions of functional tonal harmony are not explicitly encoded in *Improv RNN*, it is able to recognize basic triadic and diatonic hierarchical weightings that correspond to those conventions. This is also true for 8th notes, but not for 16th notes, indicating that harmonic interpretation depends on rhythmic context.

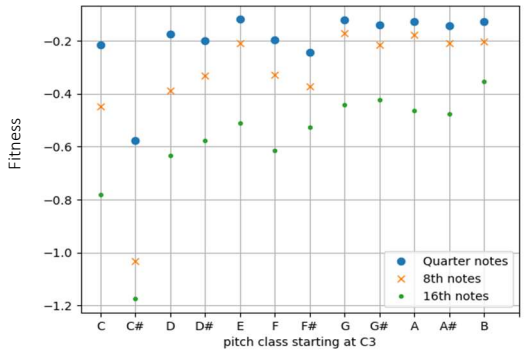

Figure 2 Fitness of pitches in the harmonic context of a C Major 7 chord (C3, E3, G3, B3).

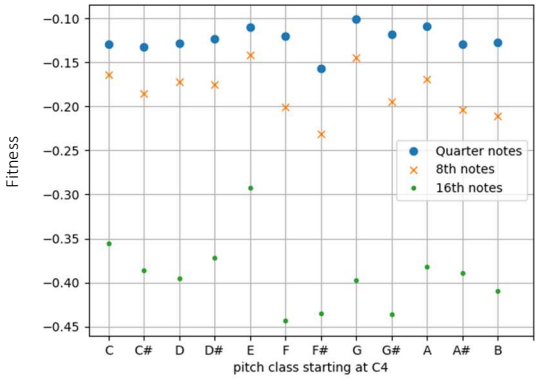

Figure 3 Fitness of pitches in the harmonic context of a C Major 7 chord (C3, E3, G3, B3). This is a continuation of Figure 2.

Looking at the fitness as a function of the length of the random note sequence provides another means of comparison (see Figure 4). The results show that the longer the sequence provided to the RNN, the more consistent its fitness.

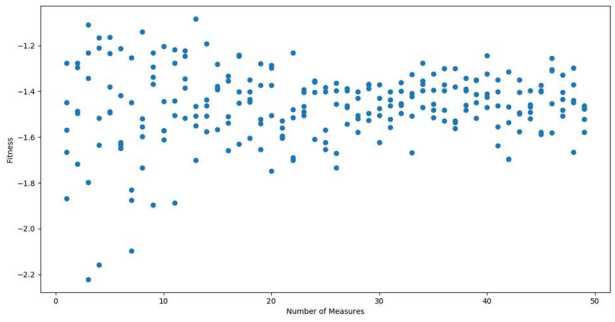

Figure 4 Increasing length gives more consistent fitness

The *Improv RNN* is able to converge to a stable total fitness for each population within 30 generations (see Figure 5). A major flaw of the *Improv RNN* is that because it prefers certain notes, convergence in practice means only those notes will be played. Generations after 14 become incredibly repetitive, and by generation 40 the population consists almost entirely of whole notes on the highest fitness pitches. Mitigating such behavior is left for future work.

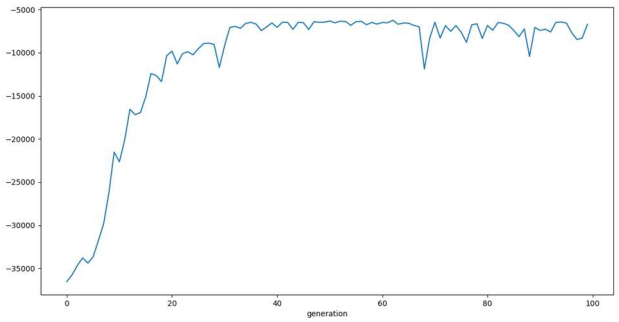

Figure 5 Sum of fitness of all measures in the measure population over time in Improv RNN training process.

Evaluating the Human Fitness Function

We now present analysis of human fitness feedback given during training. We found that roughly 10 generations with 32 measures and 48 phrases are needed before the first "golden" generation appears. This matches the results re ported by Biles (1994). The analysis in Table 1 shows the feedback that was gathered over one training session. In this instance, 17 measures out of the 32 total were given feedback. Note that not all measures in the population receive fitness feedback all the time, which makes convergence slower.

Count	Min	Max	Mean	Std. dv.	Mode
1	-1	-1	-1	0	-1
12	-6	4	-1.66	4.55	1
4	1	3	2.5	0.86	3
7	1	4	2.71	1.48	1
6	1	3	2	1	1
1	1	1	1	0	1
2	-1	1	0	1	1
3	1	2	1.66	0.47	2
7	-1	5	3.57	2.32	5
1	1	1	1	0	1
2	-1	-1	-1	0	-1
6	-2	2	0	1.73	1
7	-5	2	-3	3.16	2
35	-5	8	2.71	4.28	3
10	-2	6	3	3.68	6
4	-2	-1	-1.5	0.5	-1
14	-1	4	1.42	1.87	4

Table 1 Feedback to measures over one training session. Count represents how many times a measure received feedback. Min through mode represent statistics about the feedback scores.

This is because fitness is unchanged if neither ‘g’ nor ‘b’ is pressed by the human trainer. This is in contrast the RNNs, which provide feedback at every iteration. Each row in Table 1 represents one of those 17 measures. The results show that in many cases, feedback for a particular measure is inconsistent. For example, the second row shows that one measure was given feedback 12 times. At one point, it was given a score of -6, and another time 4.

This could mean, for example, that the measure got 6 "bad" one time (-6), and another time got 5 "good" and 1 "bad" another time (5 - 1 = 4). There are many potential explanations for this inconsistency. The value of a particular measure is not inherent, instead, it depends on the music that preceded it, which is part of its context. Different antecedents will affect the perceived value of the same consequent. Attention is another potential factor: in one instance the participant may focus on rhythm while in another instance (of the same measure) she may focus on melodic contour. Participants may be excited and interested at some times and tired or bored at others, which may influence the fitness rating given to the measure. Our findings support the notion that human listeners do not have a fixed criterion for fitness like the RNN does. The inconsistency of human feedback makes convergence to a result with high fitness difficult. This is not to say that human training doesn't work, but rather that it reaches a point where the piece stops improving.

Tuning of Parameters

The most important of the aforementioned parameters are smallest rhythmic interval and allowed pitches. The other parameters, measure population size, phrase population size, number of measures per phrase, and initial note distribution also influenced results.

Smallest rhythmic interval defines the resolution of one gene in the measure-level population. We found that 16th note durations at tempos above 120 bpm were difficult to judge in the human-feedback condition. This was especially true for the first few generations, which contained large intervals between pitches and non-repeating rhythms. We found that training was easier with a minimum rhythmic interval of an 8th note. Conversely, the RNN does not have the same kind of perceptual limitations when it comes to processing rapid information. Thus, the choice of the smallest rhythmic interval is based on aesthetic preference, and on the training data used when training the RNN.

Lists of allowed pitches define which pitches are permitted to be played over a given chord. Recall that numbers in the measure-level population are not MIDI notes themselves, but are numbers from 1 to 15 representing an index of the list of allowed notes in a chord. One might let the allowed pitches to include all of those in a desired key. However, one can further allow or restrict the lists of allowed pitches according to aesthetic preferences.

In practice, the size of a population is affected by the training condition. In the human-feedback condition, we found that evaluating more than 200 measures in a generation was tedious. Computers have no such constraints, and therefore can entertain larger initial measure and phrase populations, which lead to higher eventual fitness.

The number of measures per phrase is mostly a function of the chord progression. While a chord progression cannot last longer than a phrase, one can make the number of measures per phrase longer than one's chord progression. Longer phrases thus allow different configurations of measures and chord progressions, which allows the RNN to evaluate and influence relationships between groups on higher hierarchical levels.

Analysis of the Resulting Populations

We analyzed one *Improv RNN* training process to determine which note durations, pitches, and measures out of a population it prefers. For this analysis, we used the *Improv RNN* for 14 generations, which took about 112 minutes. This length was chosen because it was close to the amount of time used by the human trainers.

First, we looked at preferred note durations. We counted the occurrence of each duration type in each generation and plotted them against each other. The result is shown in Figure 6. The network preferred 8th notes, since they are the most prevalent over the training process and survived more than any other into the final population. However, the total number of notes played decreases and the occurrence of longer notes increases over time.

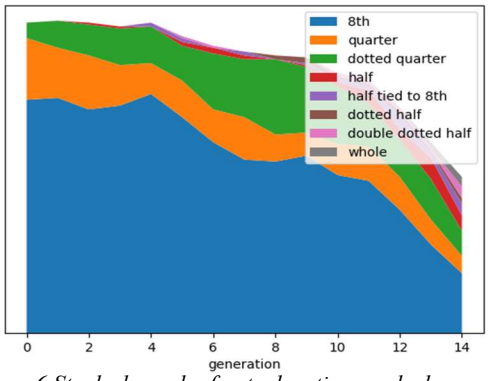

Figure 6 Stacked graph of note duration used when evolving with Improv RNN

Second, we analyzed the algorithm's usage of different pitches. The output of the algorithm can either be a rest (0), a note (1-14), or a sustain (15). Usage of each is counted and plotted for each generation (see Figure 7). Over time, the training process converged towards fewer rests and more sustained notes, and the latter became increasingly prevalent as compared with new notes. Also notable is the increase in notes 8 and 10, which in all of our lists of allowed pitches

map to the root and third of the chord one octave up. This reinforces our finding that *Improv RNN* prefers pitches that conventionally have harmonic weight.

Finally, we analyzed the measures used in each phrase of the algorithm output. Figure 8 shows a stacked plot of the prevalence of each measure in each of the algorithm's generations. While the figure doesn't represent the specific content of measures, it does show that most measures get relatively equitable usage in the population. This is surprising because during the human fitness training, some measures dominated the population (see). One explanation would be that the RNN is trained on a large and diverse corpus.

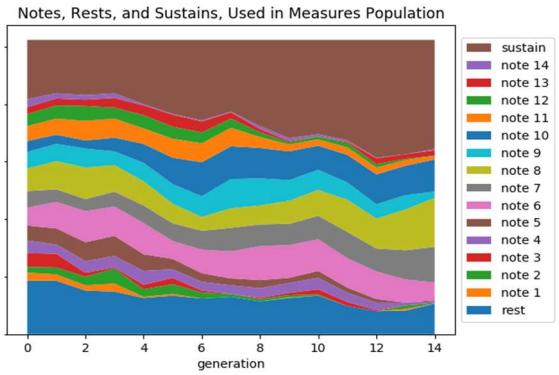

Figure 7 Stacked graph of note type usage for each measure in the population for the Improv RNN training process

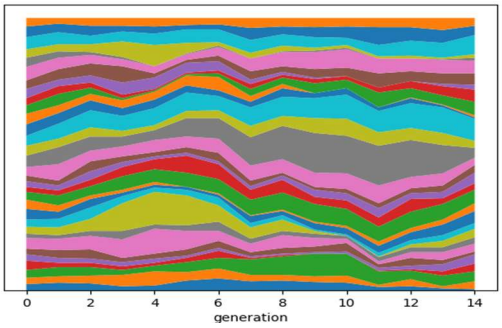

Figure 8 Stacked graph of measures used in the phrase population over generations for the Improv RNN training process

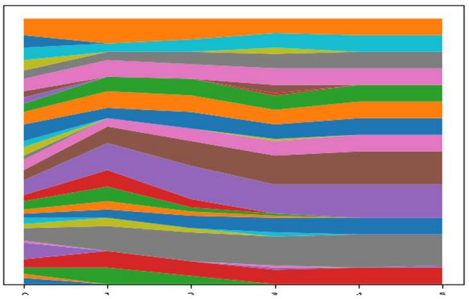

Figure 9 Stacked graph of measures used in the phrase population over generations for the human training process

Conclusion and Future Work

We show that an RNN designed for music generation can be used as a fitness function for a genetic algorithm that produces monophonic solos over a given chord progression. We conducted training sessions with human participants in order to compare and quantify some of the differences between human-feedback and RNN fitness functions. We found that the inconsistency of human feedback creates problems in regard to the convergence of solutions. Additionally, the process of manually providing feedback is tedious. Our results suggest that certain types of recurrent neural networks can address these issues, and thus should be explored in future research. Training *Improv RNN* on a large, well labeled, dataset of a specific genre or composer would be useful in exploring whether RNNs can fit closely to data with less variation, such as four-part harmony. Also of interest is how manipulation of the aforementioned parameters, such as length and allowed pitches, affects the musical characteristics of results. More rigorous experiments comparing human evaluations and RNN evaluations will provide useful feedback for designing better RNNs. Research that illuminates how these characteristics are perceived by human listeners will enable more thoughtful revisions of the algorithm. An interdisciplinary approach that draws from computer science, cognitive psychology and music theory will be helpful in this effort.

Links

You can listen to raw outputs as well as compositions made using http://petermitrano.github.io/plonky/.

The full source code for replicating our results and analysis is available at https://github.com/Arthurlockman/plonky.

Acknowledgements

Thanks to Al Biles for the inspiration for the project, and to the Magenta team at Google for making their incredible work open-source.

References

Bickerman, G., Bosley, S., Swire, P., & Keller, R. M. (2010). Learning to Create Jazz Melodies Using Deep Belief Nets. *International Conference on Computational Creativity*, 228–237.

Biles, J. A. (1994). GenJam: A Genetic Algorithm for Generating Jazz Solos. International Computer Music Conference, 131–137.

Chen, C.-C. J., & Miikkulainen, R. (2001). Creating Melodies with Evolving Recurrent Neural Networks. In Proceedings of the 2001 International Joint Conference on Neural Networks.

Geoffrey E. Hinton, D. E. Rumlhart, & Ronald J. Williams. (1988). Learning representations by back-propagating errors. Cognitive Modeling, 5(3).

Gibson, P. M., & Byrne, J. A. (1991). NEUROGEN, musical composition using genetic algorithms and cooperating neural networks. In 1991 Second International Conference on Artificial Neural Networks (pp. 309–313).

Gillick, J., Tang, K., & Keller, R. M. (2010). Machine Learning of Jazz Grammers. Computer Music Journal, 56–66.

Google Magenta. (n.d.). Retrieved March 9, 2017, from https://magenta.tensorflow.org/

Herremans, D., & Chew, E. (2016). MorpheuS: Automatic music generation with recurrent pattern constraints and tension profiles. In IEEE TENCON. Singapore: IEEE.

Keller, R. M., & Morrison, D. R. (2007). A Grammatical Approach to Automatic Improvisation. Proceedings of the Fourth Sound and Music Computing Conference.

Krumhansl, C.L. (1979). The psychological representation of musical pitch in a tonal context. *Cognitive Psychology*, *11*, 346-374.

Loughran, R., & O'Neill, M. (2016). The Popular Critic: Evolving Melodies with Popularity Driven Fitness. The Fourth International Workshop on Musical Metacreation.

Sheikholharam, P., & Teshnehlab, M. (2008). Music Composition Using Combination of Genetic Algorithms and Recurrent Neural Networks. In 2008 Eighth International Conference on Hybrid Intelligent Systems (pp. 350–355). https://doi.org/10.1109/HIS.2008.46

Simon, I. (2017, January 30). improv_rnn README.md. Retrieved March 9, 2017, from https://github.com/tensorflow/magenta/blob/master/magenta/models/improv_rnn/README.md

Waite, E. (2016, July 15). Generating Long-Term Structure in Songs and Stories. Retrieved March 9, 2017, from https://magenta.tensorflow.org/2016/07/15/lookback-rnn-attention-rnn/