Octave: A Generative Music Engine using Genetic Algorithms and Markov Matrices

Abhiram Kothapalli

Abstract

1 Introduction

Generative music, namely music that is algorithmically composed, has been explored quite thoroughly by numerous different disiplanes. In the domain of machine learning past works have used recurrent neural networks for their advantages with time based learning [1, 2, 5], genetic algorithms [3, 7], and Markov matrices [4]. Despite numerous efforts algorithmically generated music still has not achieved the quality and creativity of human composers. This work, while it does not break new grounds in the quality of the output compositions, explores a fresh approach to creating music by using genetic algorithms to evolve Markov matrices representing note transition probabilities. The key insight lies in combining both Markov matrices and genetic algorithms to create a novel and unexplored approach to generative music. We are able to achieve this sort of combination by redefining the genetic operations and fitness functions found in genetic algorithms to fit Markov matrices as population elements.

Previous Work The field of generative music is almost entirely dominated by recurrent neural networks in the machine learning domain. Solutions usually take some input musical string and attempt to learn the implicit patterns in note progression with varying degrees of success. Google's open source Magenta project [8] which uses deep learning to generate artwork and music is perhaps the most up to date and robust recurrent neural network used for this end. While Magenta has been able to produce interesting compositions they are limited in length to around 30 seconds and requires a large dataset of input sequences. This makes generating tunes and melodies on the fly cumbersome. Genetic algoritms have also been explored my multiple works but are usually limited by a weak or unrefined fitness function. Some past genetic algorithms have suggested using the human ear as a fitness function [7]. While this provides the highest quality measure of fitness it is not scalable and cannot function outside of trivial population sizes and iterations. Other fitness functions have attempted to use tonal rules [6] but these are limited in scope and cannot control the quality of overall flow and rhythm. Finally Markov matrices are one of the oldest methods and simply study the most likely probability distribution over a large data set for the next note conditioned on a previous note [4]. While Markov matrices are simple and much faster to compute than recurrent neural networks they cannot keep track of past data very well and therefore do not produce very interesting compositions when used alone.

Contribution We are the first to produce a novel combination of previous algorithms by evolving Markov matrices using genetic algorithms. This allows us to combine the speed and efficiency of generating Markov matrices with the probabilistic power of generative algorithms to select the most optimal matrix for a set of conditions. Unlike most works in the domain, our proposed algorithm does not use any seed training compositions. Instead we build our compositions from scratch using a robust fitness function using simple musical guidelines. Additionally we create an algorithm that is computationally simple to allow music to be generated cheaply and repeatedly.

2 The Generative Algorithm

The music composition algorithm works by genetically evolving Markov matrices that represent note transition probabilities.

2.1 Markov Matrix Population Element

The generative algorithm uses order 1 Markov matrices to represent note transition probabilities. Each row represents the previous note and the elements of that row represent probability distribution for the next note conditioned on the previous note. We also include a row and column to represent a pause. We can simulate the progression of notes using these probabilities to create a musical piece. Unlike standard genetic algorithms which evolve fixed segments of notes, this model allows us to define and evolve infinitely long musical pieces.

Order N Markov Matrices We additionally explore the possibility of order N Markov matrices. In this context an N Order matrix represents the probability distribution conditioned on the last N notes. However an order N Markov matrix requires N^2 rows making it computationally infeasible outside of some trivial N.

2.2 Genetic Evolution Algorithm

A genetic algorithm functions by repeatedly taking a population of candidates, selecting a subset of the population using some fitness function, and finally performing genetic operations such as crossover and mutate to create a new population. This process can be summarized as below:

1. Initalize: Randomly create a population by generating N candidates
2. Select: Select an optimal subset of population P_k by picking $S \subset P_k$ based on some fitness function $f(P_{ki})$
3. Genetic Operations
   (a) Crossover: Pick two elements $s_1, s_2 \in S$ such that elements with higher fitness have a higher probability of being picked. Merge these elements using some function $crossover(s_1, s_2)$ and place the output element in population P_{k+1}.
   (b) Mutate: Modify element $s \in P_{k+1}$ with a small probability γ
4. Iterate: Continue iterating the population by repeating steps 2 and 3 until the population fitness converges.

We redefine the genetic operations and fitness function to fit the constraints of Markov matrices as population elements.

2.2.1 Genetic Operations on Matrices

Crossover While there are several approaches to merging two Markov matrices, We chose to merge s_1 and s_2 by picking a random split point, j, among the rows and copying rows above j from s_1 and rows j and below from s_2. Intuitively this means that we inherit the probability distributions for some notes from parent 1 and the distributions for the remaining notes from parent 2.

Algorithm 1 Crossover

```
Initalize randomly selected split point, j
Initalize parents $s_1, s_2$
while i < number of rows do
    if $i < j$ then
        $child_i \leftarrow s_{1i}$
    if $i \geq j$ then
        $child_i \leftarrow s_{2i}$
    $i \leftarrow i + 1$
Return child
```

Mutation There are also several options for mutating matrices. We simply modify an element in the matrix with some set mutation probability, γ.

Algorithm 2 Mutation

```
Assume some $\gamma$ has been set
while i < number of rows do
    while j < number of columns do
        if Probability $\gamma$ is satisfied then
            $matrix[i][j] \leftarrow$ random new probability
```

2.2.2 Fitness Function on Matrices

As with any genetic algorithm the fitness function is crucial to creating Markov matrices that represent satisfying music. In fact, creating an effective fitness function to judge musical compositions is an active area in research. In an attempt to preserve creativity and allow a wide variety of results, we aimed to create a simple yet effective fitness function. A simple fitness function essentially needs to capture and judge adherence to the fundamental rules of pleasing music. To achieve this end, we identify several key characteristics of pleasing music:

1. Ascending and descending note sequences
2. Occasional pauses
3. A balance between long and short notes
4. Unexpected jumps and variations

Implementing these rules individually is easy. We can just give larger fitness scores to matrices that have transition probabilities adhering to these sorts of rules. However these rules when used individually as fitness functions lead to uninteresting and repetitive melodies, due to their simplicity. We can create more interesting rules by creating a new fitness score resulting in a weighted linear combination of simpler fitness scores. For example a more robust fitness function can value frequent pauses and unexpected jumps placing a higher emphasis on the latter. This allows us to keep the overall simplicity of assessing the fitness of our matrices while still producing interesting compositions. If we have simplistic fitness functions $f_1, f_2, f_3, ..., f_k$ and weights $w_1, w_2, w_3, ..., w_k$ for these fitness functions, our robust fitness function, f_r, can be summarized with the following representation.

$$f_r(m) = \sum_{i=1}^{k} w_i * f_i(m)$$

3 Implementation and Evaluation

We have created a working implementation of the proposed generative algorithm in order to experimentally evaluate the musical quality based on different parameters and fitness functions. The breakdown of the implementation can be described by its component files:

1. generator.py: Contains the genetic algorithm process. Takes in population element constructor, crossover, mutate and fitness function as an argument. Also takes in number of generations, number of elements per population and exclusivity function.
2. markov.py: contains definitions for how to construct, crossover, and mutate a markov population element. Also contains the fitness function rule based on fitness.py

3. fitness.py: Contains various fitness functions that can be used and combined in markov.py playmarkov.py: generates a Markov matrix with specifications set in markov.py or takes an argument to an existing markov matrix. Plays the resulting Markov matrix.

The projects source code and supporting files can be found at the following links. We can subjectively assess of the success of the proposed algorithm by listening to the provided sound samples.

- Source: https://github.com/abhiramkothapalli/Octave.
- Sample Compositions: https://github.com/abhiramkothapalli/Octave/tree/master/samples.
- Sample Matrices: https://github.com/abhiramkothapalli/Octave/tree/master/matrices

3.1 Parameter Analysis

3.1.1 Varying Mutation Probability

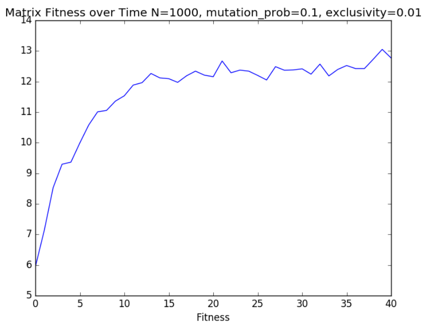

Figure 1: Mutation Probability 0.1

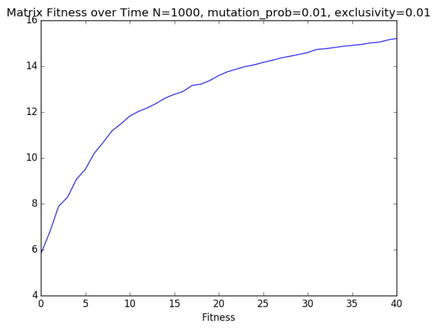

Figure 2: Mutation Probability 0.01

As seen in the figures above a reasonably small mutation probability converges more stably and converges to a higher population fitness. The second configuration is able to produce music that fits the prescribed rules better but does not necessarily sound better. This is a good example of how weak tuning, in fact, allows more space for creativity giving us more interesting compositions.

3.1.2 Varying Population Size

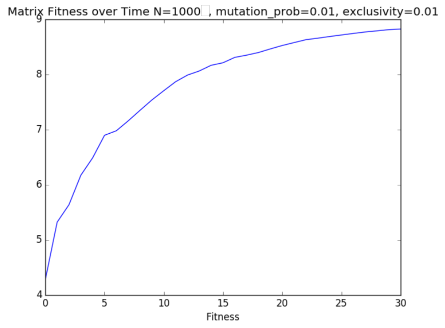

Figure 3: Mutation Probability 1000

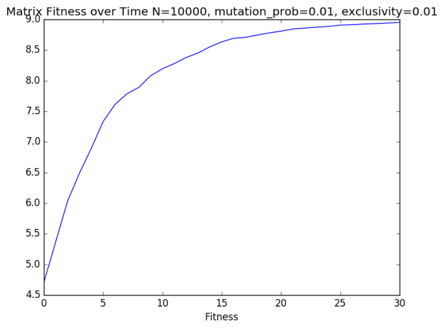

Figure 4: N = 10000

Upon close inspection we can tell that a population size of 10000 converges faster but both end up with the same fitness score. This show's that over time the number of population elements so long as it's reasonably high is insignificant. This implies that we can get fairly optimized compositions with only a relatively small population size.

4 Future Work

5 Conclusion

References

[1] Memo Akten. Realtime control of sequence generation with character based long short term memory recurrent neural networks. *Memory*, 9(8):1735–1780.

[2] C. C. J. Chen and R. Miikkulainen. Creating melodies with evolving recurrent neural networks. In *Neural Networks, 2001. Proceedings. IJCNN '01. International Joint Conference on*, volume 3, pages 2241–2246 vol.3, 2001.

[3] Arne Eigenfeldt. Generative music for live musicians: An unnatural selection. 2015.

[4] Jose D Fernández and Francisco Vico. Ai methods in algorithmic composition: A comprehensive survey. *Journal of Artificial Intelligence Research*, 48:513–582, 2013.

[5] Alex Graves. Generating sequences with recurrent neural networks. *CoRR*, abs/1308.0850, 2013.

[6] Dragan Matić. A genetic algorithm for composing music. *Yugoslav Journal of Operations Research ISSN: 0354-0243 EISSN: 2334-6043*, 20(1), 2013.

[7] Eduardo Miranda. *Composing music with computers.* CRC Press, 2001.

[8] Tillman Weyde. Perception, cognition, and generation of music: a case for human-like understanding of temporal structures.