Chiptune Genetic Algorithm Final Documentation

Ahmad Azeez, T00686506

Meenakshi Rajesh, T00596789

COMP3710_01 - Applied Artificial Intelligence

Thompson Rivers University

Dr. Joseph Alexander Brown

December 3rd, 2024

Table of Contents

Problem Description

Chiptune music is recognizable by its striking 8-bit aural characteristic. It is often associated with the classic gaming era, whereby technical limitations led to the production of simple yet memorable musical pieces. This genre can be typified by using waveforms such as square, triangle, and sawtooth that render coarse, synthesized sounds. This evokes nostalgia and represents some parts of retro gaming culture.

The project used genetic algorithms for chiptune creation. Applying concepts like crossover, mutation, and evaluation of fitness, the algorithm evolves a population of audio samples over a number of generations to achieve an output that fits the input audio.

Chiptune music is well-suited for this method for several reasons. Chiptune music is realized with simple waveforms and basic synthesis techniques, making it easily manipulable by a genetic algorithm without any of the complexities found in modern music production. This simplicity in sound allows the algorithm to focus on the primary musical elements: pitch, timing, and rhythm. The somewhat expected structure of chiptune music, with its usually repetitive patterns, fixed note lengths, and clear melody, makes it a good candidate for evolutionary methods. Genetic algorithms can efficiently tune these elements to optimize specific goals, like fitting the melody and rhythm of a given audio track.

Method

This research project uses a genetic algorithm to generate chiptune music from an input audio or MIDI file. The algorithm evolves, over time, a population of audio representations with the goal of improving melodic and rhythmic fitness relative to the input, all while adhering to aesthetic constraints typical of 8-bit chiptune music.

Initialization

The process begins with reading a MIDI audio file. Each note in the MIDI file is then logged as a pair of note and duration where note is the corresponding MIDI note number (for instance, 67 for G4). A population of such note-duration pairs is randomly initialized, constrained by rules inherent to the chiptune music, such as using square waveforms to generate sound.

- **Representative sample:** Each chromosome is a sequence of tuples (note, duration) representing a melody and rhythm. For example, [(67, 0.5), (71, 0.25), (64, 0.75)], where the 67 is the MIDI note for G4 and 0.5 indicates the duration of the note in seconds. An initial population is created by sampling a range of common MIDI notes and rhythm lengths, thereby ensuring a certain amount of diversity.

Fitness Test

The fitness function evaluates how similar each audio sample is to the original input, in terms of melody and rhythm.

- It compares the generated MIDI notes with the original input notes.
- Measures how well the note durations produced correspond to the inputs.
- Uses square waves but constrains the complexity to retain characteristics of 8-bit sound. For example, if the target melody's length is 12, a score out of 12 is given, 12 meaning a perfect fit of both note and duration to the input audio.

Selection and Crossover

After calculating fitness values, the high-fitness chromosomes are chosen as parents through tournament selection.

- **Selection Tournament:** It uses randomly selected subsets of the population, the chromosome with the highest fitness in each subset is declared a parent.
- **Cross-over Operation:** Parent chromosomes exchange portions of their note-duration pairs to form offspring. For example:
    - Parent 1: [(67, 0.5), (71, 0.25)].
    - Parent 2: [(64, 0.75), (60, 0.5)].
    - Offspring: [(67, 0.5), (60, 0.5)].

Mutation

Random mutations are also introduced in some small percentage of offsprings (depending on the mutation rate) to increase the diversity of the population.

- **Mutation Operations:** Modulates the MIDI note number by a fixed range. For example, replacing note 67 with 68.
- **Duration Manipulation:** Randomly increases or decreases the note duration within acceptable limits. For example, a decrease from 0.6 seconds to 0.5 seconds.

Evolution and Conclusion

The genetic algorithm progressively develops the population through a continuous cycle of fitness assessment, selection, crossover, and mutation until one of the high-fitness chromosomes generated after each generation matches the input MIDI representation exactly. The final chromosome is converted back into an audio using Python-based waveform synthesis. Only square waveforms are used in the generation of the 8-bit audio file.

Works such as "*Genetic Algorithm in Python generates Music*" by Kie Codes [1] and "*How I made a genetic algorithm that generates music*" by Taimur Shaikh [2] were of great help for the development of this project. These works helped with figuring out how to create a genetic algorithm for audio in Python.

Materials

The materials for this project include resources on 8 bit audio, generative AI methods along with essential Python libraries for audio and MIDI manipulation.

Information Resources

- **Input Audio and MIDIs:** All MIDI files were obtained from bitmidi.com [7], 'MissionImpossible.mid' [3], 'Mortal Kombat - Theme.mid' [4], and "Theme-From-'The-Pink-Panther'-2.mid" [5]. These were needed as input to the genetic algorithm.
- **Sound Processing Basics:** Knowledge on waveforms, samples, sample rate, audio amplitude, and frequency.
- **Generative AI:** Chromosomes, crossover, mutation, and fitness function.

Software Tools

- **Python:** Libraries such as 'Numpy', 'PyDub', and 'mido' were used for audio and MIDI processing and manipulation. Other libraries like 'random', 'typing' (for type safety), and 'keyboard' were used for quality of life features. All these open source libraries are available through Python's pip package manager.
- **FL Studio:** This was used to isolate the melody in the MIDI, due to the monophonic nature of the genetic algorithm. It was also used to produce the audio of each collected MIDI.

Mentorship and Guidance

- **Technical guidance**: Throughout the project , Dr. Brown provided us valuable insights on chiptune music. He introduced us to some chiptune music from old retro games, which helped us to understand the evolution and distinctive characteristics of 8 bit audio over time. He offered advice on how many generations it might take for the genetic algorithm to specifically converge to the input audio. This helped us to adjust mutation rates and numbers of population accordingly to get an accurate match of the input melody.

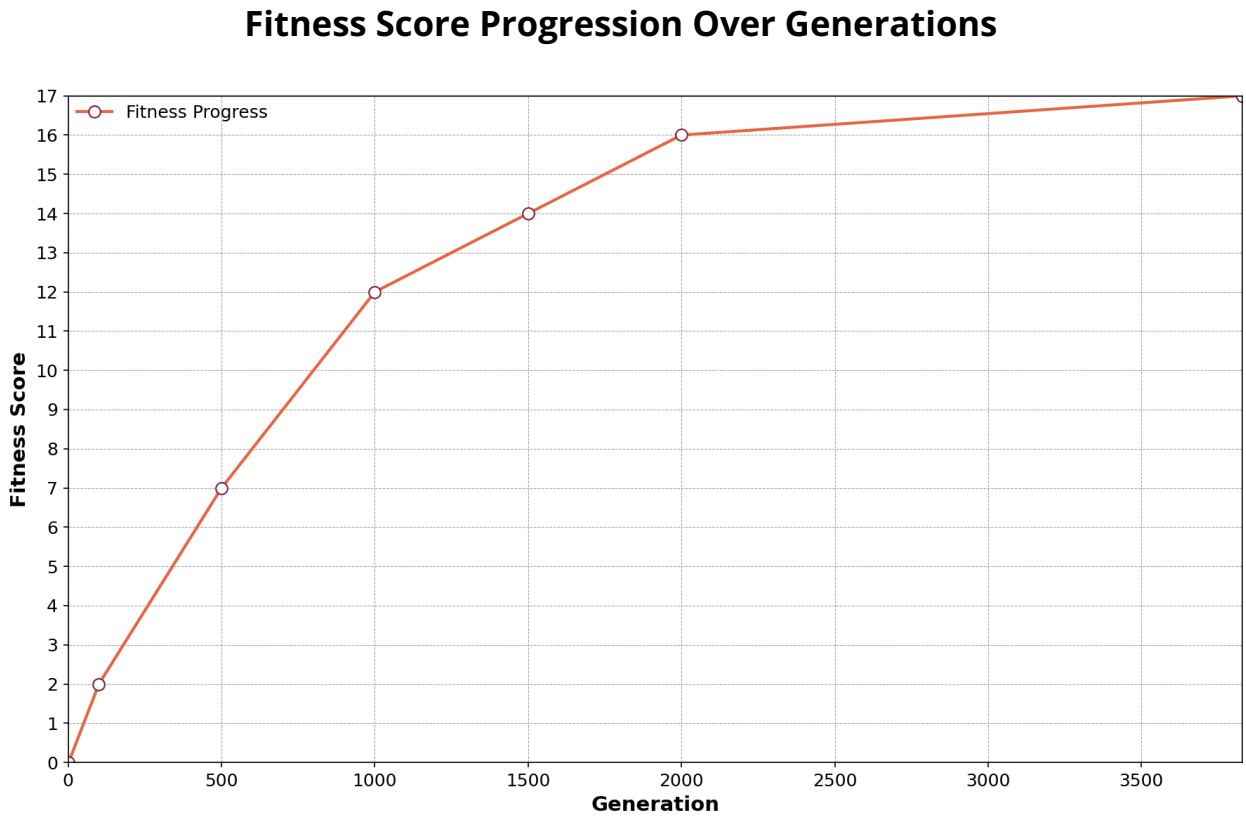
Evaluation

The genetic algorithm used in this project was evaluated based on its ability to generate melodies that closely matched a target melody, using fitness scores as a metric for similarity. The fitness score was calculated by comparing each note and its timing in the generated sequence to the corresponding note and timing in the target melody. A higher fitness score indicates greater similarity, and the max fitness score is determined by the length of the target melody MIDI array.

Success Metrics

The main success metric was the improvement of the fitness score across generations. The algorithm's performance was observed over multiple generations to track convergence toward the target melody, with a 50% mutation rate and a population of 1,000 (the size of each chromosome is determined by the size of the target melody array).

Results



The above graph shows how close each generation got to the "best fitness score" (17 in this case).

- **1 Generation:** The initial fitness score was 0, indicating that the generated melody had no resemblance to the target melody.
- **100 Generations:** The fitness score improved slightly to 2, which indicated early-stage evolution.
- **500 Generations:** The fitness score reached 7, showing noticeable improvement. The algorithm was beginning to better align the melody, though further refinement was clearly needed.
- **1000 Generations:** The fitness score rose to 12. The algorithm had started producing melodies with greater accuracy.
- **1500 Generations:** The fitness score reached 14.
- **2000 Generations:** The fitness score increased to 16. The melodies were closely matching the target sequence.
- **3828 Generations (Best Fit Found):** The best fitness score of 17 was achieved. This was the maximum fitness score, indicating that the genetic algorithm had successfully converged to a melody very close and identical to the target, but with a chiptune overhaul as intended.

An Interesting Observation

The time it takes for the last fitness score to be reached seems to be the longest out of the rest. It appears that each increase in the fitness score took progressively longer to achieve. As the generations advanced, the algorithm required more iterations to make improvements, indicating that the population was approaching a local optimum and further refinements became more challenging. This provides an explanation for the prolonged time to get to the final fitness score.

The mean and standard deviation of the fitness scores were also calculated:

- **Mean Fitness Score:** From the example above, the fitness score showed a steady increase. Starting with an average score of 0% in the initial generation, it progressed to intermediate values like 2, 7, and 12 at 100, 500, and 1000 generations, respectively. By the final stages, the fitness score reached a mean value of approximately 9.71, showing the algorithm's ability to refine solutions closer to the target melody.

- **Standard Deviation:** Decreased over time, showing a more homogenous population as the algorithm converged.

Limitations

While developing the algorithm, it occasionally stagnated and the fitness score would hardly move, requiring adjustments in the mutation rate or population size to add diversity. Also, the timing values for some of the notes had to be rounded up to fewer decimal places because the decimal numbers of the original timing values were long and caused the algorithm to take a significant amount of time to match the target melody.

Intially, raw audio was attempted to be used, but because it is a continuous waveform and does not have a note based representation, it was hard for the genetic algorithm to provide solutions. A python program that converted the audio into MIDI format was created, but this was also unsuccessful as it only worked for one audio file, one containing the ‘Mary Had a Little Lamb’ melody, due to the fact that the program was created with only that melody sample in mind.

Eventually, a separate python program that read MIDI files directly was created since the algorithm successfully generated a chiptune version of any monophonic MIDI file provided. This means the algorithm can only accept MIDI files, except for the audio file of ‘Mary Had a Little Lamb’ apparently.

Gallery

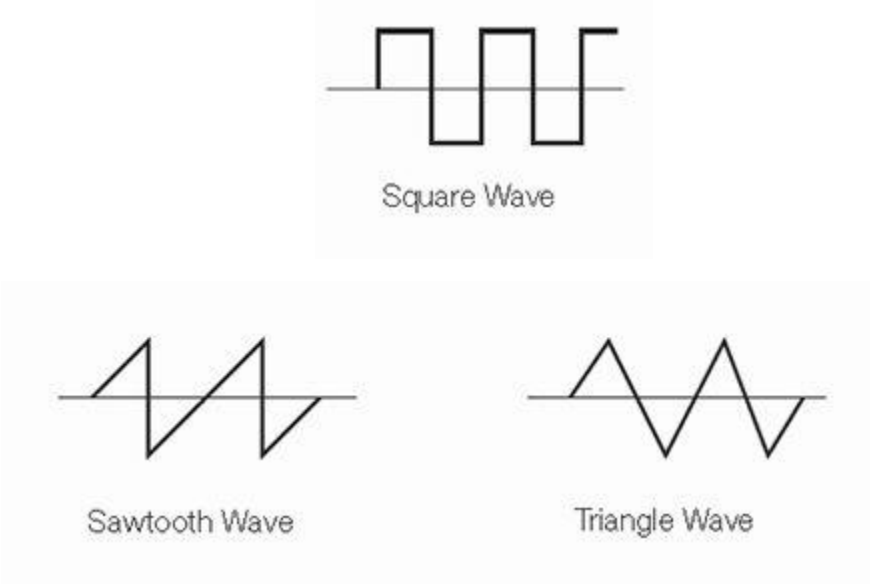
The goal of this project was to generate music that evokes the nostalgic essence of 8-bit sounds from classic retro video games, paying homage to an era when sound technology was both a constraint and a catalyst for creativity.

The music generated by this project is a mix of artificial intelligence and creativity. Each melody evolves over multiple generations through a genetic algorithm. The project uses a selection of random initial melodies which gradually evolve toward a target melody through processes of mutation and crossover.

Each generation represents an iteration of improvement, where the fitness score shows how closely the generated melody matches the target.

MIDI was chosen for this project because it was a very structured and straightforward representation of music, far more suitable for manipulation via a genetic algorithm. This is one of the major characteristics of MIDI files (.mid), containing discrete pairs of notes and their durations. Unlike raw audio, which consists of intricate waveforms, MIDI represents music in a far more accessible manner with regard to parameters such as pitch, rhythm, and tempo. It is lighter compared to audio files, which allows fast experimentation and optimization within a genetic algorithm.

Square waveforms are one of the defining characteristics of chiptune music closely associated with retro gaming. Using square waves is really important to keep the authenticity of the chiptune style, as they create a rather distinct, coarse, and sharp sound that is reminiscent of classic gaming consoles. Square waves in this particular project mean that the result fits stylistic and sonic constraints of 8-bit music and are easy to be generated and manipulated which makes them a nice fit with an algorithmic approach.

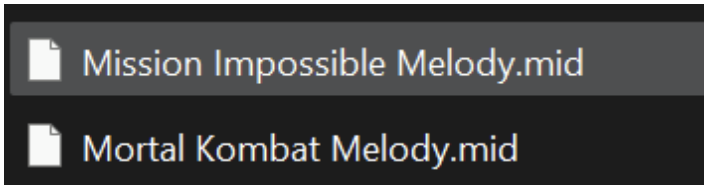


Waveform diagrams from Circuit Basics [6].

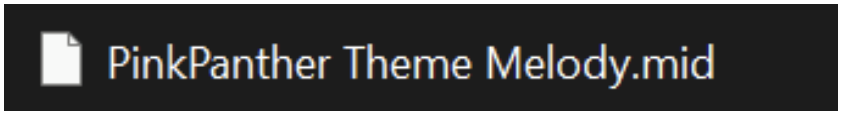
Other waveforms were used too, such as triangle waves and sawtooth waves, but since square waves were the most popular for retro games, it was used in this project. Anything more complex (in terms of the bit depth, and waveform used) than this would not be Chiptune. For example, sine waves were hardly used back then due to their complexity and the hardware limitations.

Listeners should care about this project since it represents the epitome of artificial intelligence and creative expression coming together: genetic algorithms creating music in a genre that is both nostalgic and innovative. Music created under this project might resonate with a strong relation to the past, yet it is as much an invitation to new audiences to partake in this rather different kind of musical creativity. As an AI-generated art, the project raises important questions about creativity, authorship, and the future of music production.

To test the algorithm, three different MIDI files were used. ‘Mission Impossible’ was chosen due to its recognizability and melodic “simplicity”, same for ‘Mortal Kombat’.



Then for the third MIDI file, the Pink Panther theme was chosen to see how the algorithm will function with a more “complex” melody. It was also chosen due to its recognizability.



Next Steps

There are many ways to take the project even further, such as moving from working with only small segments of notes (monophonic) to directly converting full tracks (polyphonic) into MIDI format. The upgradation of the chromosome structure into a 2D array representation could store and handle detailed information like the pitch, duration, and amplitude of the notes. This representation not only saves processing time but also allows more precise manipulation of the input audio. However, this complexity could slow down the generation times therefore a well designed genetic algorithm to accommodate these changes is needed.

Based on this idea, instead of converting an entire audio to a single track, separating different musical instruments (melody, percussion, bassline) and applying genetic algorithms to each separately and then combining each layer together would create a richer audio output.

Having users provide feedback on the generated chiptune music can also be taken into consideration and can be acheived by adding a “human” rating system.

References

[1] YouTube, "Kie Codes - Genetic Algorithm in Python generates Music," *YouTube*, Aug. 17, 2020. https://www.youtube.com/watch?v=aOsET8KapQQ. (accessed Nov. 27, 2024).

[2] T. Shaikh, "How I made a genetic algorithm that generates music," *Medium*, Apr. 17, 2021. https://medium.com/dc-csr/how-i-made-a-genetic-algorithm-that-generates-music-67b90cd0d05e. (accessed Nov. 27, 2024).

[3] "MissionImpossible.mid — Free MIDI — BitMidi," *BitMidi*, Jul. 14, 2018. https://bitmidi.com/missionimpossible-mid-1 (accessed Nov. 29, 2024).

[4] "Mortal Kombat - Theme.mid — Free MIDI — BitMidi," *BitMidi*, Jul. 14, 2018. https://bitmidi.com/mortal-kombat-theme-mid (accessed Nov. 29, 2024).

[5] "Theme-From-'The-Pink-Panther'-2.mid — Free MIDI — BitMidi," *BitMidi*, Jul. 14, 2018. https://bitmidi.com/theme-from-the-pink-panther-2-mid (accessed Nov. 29, 2024).

[6] G. L. | D. Electronics | 3, "What Are Square Wave Generators?," *Circuit Basics*, Dec. 14, 2020. https://www.circuitbasics.com/what-are-square-wave-generators/ accessed Dec. 03, 2024).

[7] "Popular MIDIs," *BitMidi*. https://bitmidi.com/ (accessed Nov. 29, 2024).