Constructing, Manipulating, Classifying and Generating Audio with Digital Signal Processing and Machine Learning

Matthew R Finch · Follow
Published in Towards Data Science
20 min read · Apr 18, 2020

Listen Share ••• More

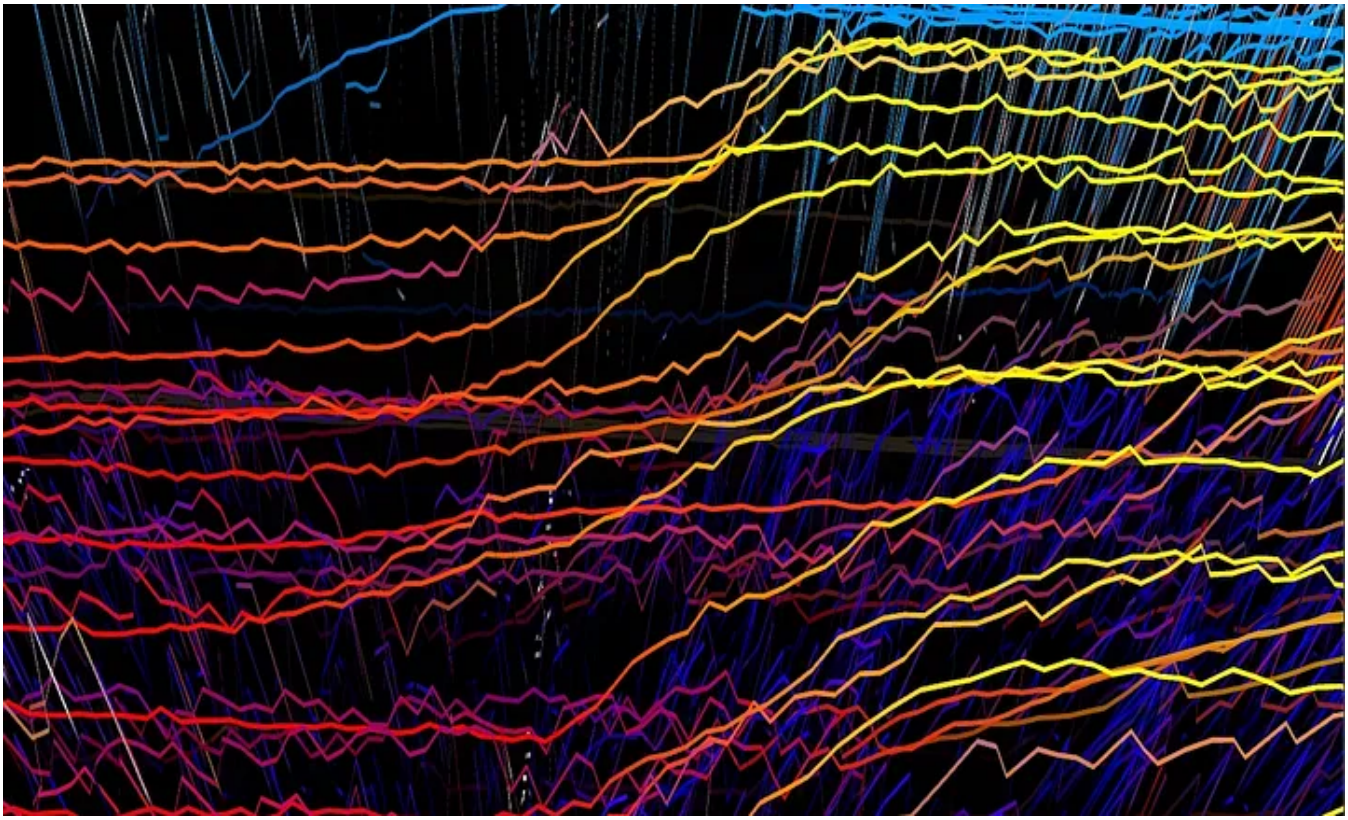


image by author

> “We live in a world where there is more and more information, and less and less meaning.”

> "And so art is everywhere, since artifice is at the very heart of reality. And so art is dead, not only because its critical transcendence is gone, but because reality itself, entirely impregnated by an aesthetic which is inseparable from its own structure, has been confused with its own image. Reality no longer has the time to take on the appearance of reality. It no longer even surpasses fiction: it captures every dream even before it takes on the appearance of a dream."

Written in 1981, both of these quotes come from Jean Baudrillard's *Simulation and Simulacra*. Adjacent is the idea of *Hyperreality;* defined as:

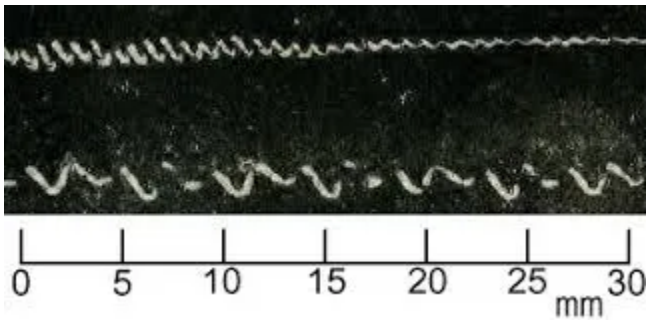
> ***Hyperreality**, in semiotics and postmodernism, is an inability of consciousness to distinguish reality from a simulation of reality, especially in technologically advanced postmodern societies*

If you Google '*What is Audio?'* — you will get a few results.

1. It originates from the latin word: '*audire'.* Which means: '*to hear'*
2. It wasn't until the late 1800's and early 1900's that the word was widely used in the english corpus. There is a slight upward trend starting a little before 1880, and then a sudden rise right before it takes off between 1900 and 1910.

google books ngram viewer, word: **audio** 1800–2019

The **Phonautograph** was first patented in France during March of 1857, by Édouard-Léon Scott de Martinville. It is the earliest known device to record sound. It would represent sound waves as lines traced onto smoke blackened paper or glass. The representation was a direct reaction from the physical phenomenon of sound waves moving through the air. Initially used to study acoustics, it was not a method of distributing sound as a media, but was used, in some cases, to detect musical pitch by transcribing a person playing a pitch on an instrument or singing and then comparing the transcription to another transcription made from a tuning fork. Using a representation to classify a representation.



É.-L. Scott, “Application for Patent №31470” dated 25 March 1857. 6 pp. including certificate and one phonautogram | source

Below is a recording made with the Phonautograph on April 9th, 1860. Believed to be Scott, himself, singing a French folk song: ‘*Au clair de la lune*’

Recording from Phonautograph, 1860, April 9th

An interesting note: when Scott made the recordings, it is said that he had no intention to have them heard by anyone, but rather to simply be an analytical tool. It wasn't until 1877 that someone thought a sound could be reproduced by a representation.

Prior to 2008, Scott's recordings were not audibly realized and the oldest recordings available were Edison's **Phonograph — yellow paraffine cylinder** captures during June 29th, 1888's concert of 4000 voices performing '*Israel in Egypt*' at the Handel Festival in London, England.

The very first in-ear device, however, was invented in the 1850s. A medical **Stethoscope.** The first stethoscope was noted in 1816. At that time it was not an in-ear device that we know now. It was a tool to amplify the sounds of the chest. Rather than using electricity, the stethoscopes of 1816, were paper funnels the doctor would place against a patient's chest. The stethoscope of 1850 wasn't too different but had become more refined as a bi-aural device; allowing the sounds from a patient's chest to enter directly to both of the doctor's ears.

In 1895 we get the first transmissions using *radio signals*. Although, the first transmission was a telegraph of the letter 'S'. So, not really dealing with sound. But in 1910, Nathaniel Baldwin invents, what we now know as the modern **Headphones**. Initially intended for the Navy's use as a method of communication, it was designed containing a mile of copper wiring connected to the operator's headband. This allowed for an analog signal to travel through the copper via electricity.

The third result you'll find when Googling *'What is Audio',* is this definition:

> *An audio signal is a representation of sound, typically using a level of electrical voltage for analog signals, and a series of binary numbers for digital signals. Audio signals have frequencies in the audio frequency range of roughly 20 to 20,000 Hz, which corresponds to the lower and upper limits of human hearing.*

For the purposes of this article, lets explore digital audio and the methods used by a machine to represent sound.

How is Sound Simulated with Digital Signal Processing?

A digital signal, like an analog signal, is a representation of sound. There are two main aspects to a digital signal that allows a machine to render a representation. These two aspects can be thought of as X and Y; Time and Amplitude, respectively: **Sample Rate** and **Bit Depth.**

Sample Rate

If you've ever opened a Digital Audio Workspace (DAW), or inspected an audio file, you will be familiar with the number 44100. If you have not, 44100Hz is the standard sampling rate for digital rendering of audio. There are different rates, such as 88200, 96000 and 196000, as well. But 44100 is a specific number and the lowest threshold of sampling a signal.

Sampling rate is the number of times per second that the machine is analyzing incoming sound or an analog signal while recording. Defined online as: '*The spatial* ***frequency*** *of the digital* ***sampling****. The reciprocal of the center-to-center distance between adjacent pixels*'. What this means is that for each sample *(1/44100 of a second)* the machine records the signal and represents each bin as a digit. Audio can also be constructed with these principles as well.

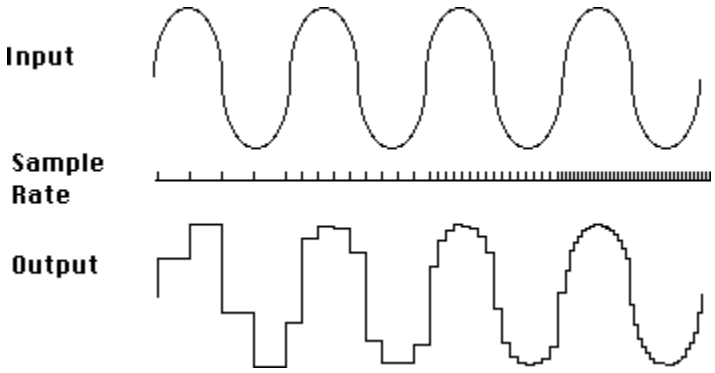


illustration of low to high (left to right) sampling rate's effect on rendering a digital representation | source

In the image above, you can see that with a low sampling rate the representation misses a lot of information in the signal; and as the sampling rate increases, produces a more accurate representation. To break it down: *the task while converting sound, or analog signals, to digital signals is to take a continuous signal and convert it into discrete values, that when put together over time, render a representation of the continuous signal that can then be converted back*. Basically, the higher the sampling rate, the more accurate and precise the machine can render. But why 44100Hz?

Nyquist's theorem of sampling states: *to achieve a sampling rate that most accurately renders the signal of a frequency, the sampling rate must be double the frequency*.

```
Sampling_Rate_Needed = frequency * 2
```

So if you have the frequency of 440Hz, then the sampling rate needed to most accurately render a digital representation of the frequency, would be a sampling rate of 880Hz. Extrapolating on that, the human range of hearing has an upper limit of, roughly, 20,000Hz. Therefore, to accurately render any digital representation of sound that humans can hear, we need a minimum of 40,000Hz sampling rate. And it seems that for safe measure, 4,100Hz were added on. The additional sampling rates are used in specific cases such as, surround sound, audio in films, music production and other hi-fidelity experiences that require more precise manipulation of the rendered signal. But to accurately represent anything a human can hear, 44100Hz is the necessary rate.

Bit Depth

Bit depth is similar to sample rate in that it is a type of resolution. But rather than rendering along time, it is rendering along amplitude. A low resolution bit depth of 1 can only render amplitude as two dynamics: On or Off. Sound or Silence. Analogous to an image in Black and White.

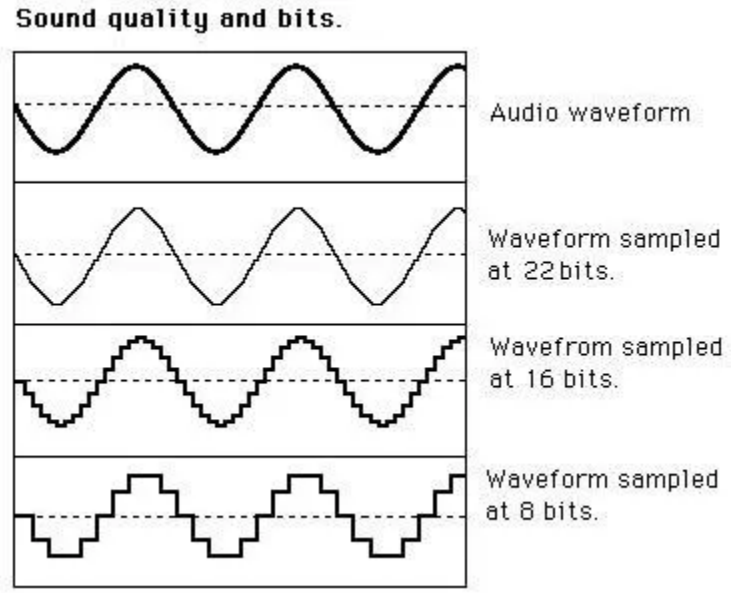


illustration of different bit depth quantization's effect on rendering a digital representation | source

Here you can see that with more bit depth, there are more discrete values available for the machine to render the representation of amplitude. We can calculate the

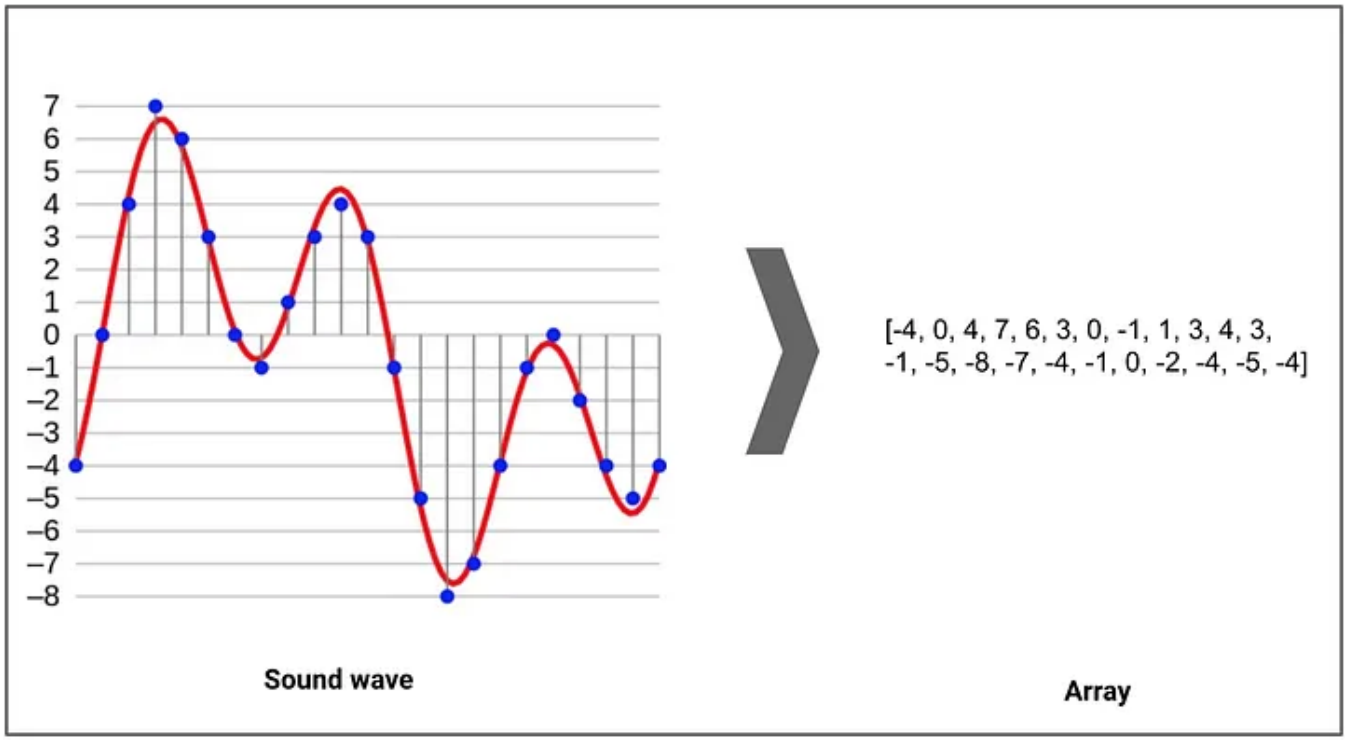
number of discrete values per bit depth with this formula:

```
2^bit_depth
```

For a depth of 8 bits, there is a range of 256 discrete values *(0–255)*. For 16 bits, there is a range of 65536 discrete values. And with 24bits, there is a range of 16777216 discrete values. Another way to conceptualize this range of discrete values is: the possible dynamic range of amplitudes available for the machine to render volume of a given sample. Using the below formula, we can understand the dynamic range of a bit depth in decibels.

```
n = bit_depth
20 * log10(2^n) = db
```

With a bit depth of 8, we have 48db in dynamic range. However, with a bit depth of 16, we have 96db; and when we factor in dithering, we can reach 115db. For a human, 140db is the limit before our ears become painfully damaged. Therefore 115db is plenty of dynamic range for most production needs when it comes to rendering sound.



basic overview of sound signal being represented into discrete digits as an array | Original © Aquegg | Wikimedia Commons

The above image shows the signal as a red line, being analyzed by the machine with blue dots and simulated with a digital rendering as an array to the right. Assuming this graph is one second, the sampling rate would be 21Hz with a bit depth of 4bits. Not sure that would sound too good.

Most music we listen to is at 44.1KHz with a 16bit depth. So next time you're listening to audio from a digital source, try to imagine all the discrete values along frequency *(44100 samples per second)* and amplitude *(65536 levels of dynamic range)* that are tightly rendered into a composite that simulates the experience of listening to a continuous acoustic sound wave.

Constructing Digital Audio

Engineering Frequencies

In my last article, Sonifying PI, I constructed an overtone series based off the frequency 3.14Hz and generated audio files from the series. Here I will go more in depth about the module's functionality that I built to accomplish this. Furthermore, I'll explain how to construct the western musical system as well as more complex, unique musical systems.

When we are thinking about what a musical pitch is, we often think of their letter names.' A, A#, G, G#' … etc. But these are only abstract representations to help us understand music as a language and navigate through the sonic landscape symbolically. 'A' represents a class of frequencies. To reference a more specific instance of 'A', we can call the pitch along with the octave number associated with its position in the series. This is called Scientific Notation. The most common instance of 'A' is 'A4'. This specific instance represents the frequency 440Hz and is in the position of the 4th octave. The current standard of tuning is based on 440Hz. Each octave position is exactly 1/2 of the octave above it and exactly double the octave below it. For example, 'A5' represents the frequency 880Hz, and 'A3' represents the frequency 220Hz. And in between each octave, there is an instance of all twelve notes in the western system. The minimum interval between two frequencies is called a minor second. A minor second is also a symbolic abstraction that holds connotations of quality and definition within the western context; but is only a representation of a numerical distance between two frequencies. There are many ways to evaluate this distance.

In the notebook below are examples of the mathematical relations necessary to develop an octave as used in the *Western Equal Tempered Twelve Tone System of Music.*

In [1]:
```
from freq_table import *
```

Navigating Octave Instances of a Pitch

In [2]:
```
# pitch_to_frequency is a dictionary i built that has all frequenc
# by calling the pitch name and the ocatve the pitch is in, you ge

# here is an example of me calling the three instances of A in oct
print(str(pitch_to_frequency.get('A')[3]) + ' Hz')
print(str(pitch_to_frequency.get('A')[4]) + ' Hz')
print(str(pitch_to_frequency.get('A')[5]) + ' Hz')
```

```
220.0 Hz
440.0 Hz
880.0 Hz
```

In [3]:
```
# here we see that a single octave is a representation of a number
one_octave = semitone_intervals.get('P8')
# here we see that A4 is a representation of the frequency 440 Hz
A_4 = pitch_to_frequency.get('A')[4]

# here is intervallic relationship between octaves
print(str(A_4 * one_octave) + ' Hz' + ' is one octave above ' + st
print('----------------440--------------------')
```

M_sys.ipynb hosted with ❤️ by GitHub view raw

notebook breaking down the mathematical factors of intervals in the western musical system

The point of this notebook it to expose a few simple truths.

1. An octave is a representation of the space between 1 and 2.
2. Dividing the octave is the division of the space between 1 and 2 into equal intervals.
3. The intervals that are equivalent to the those divisions are represented by an abstracted characterization and are unique respective to the octave they are native to — but share common factorization.

But let's use this logic and investigate some alternative musical systems.

Navigating Alternative Musical Systems

Navigating Alternative Musical Systems

```
In [1]: from freq_table import *
        from wav_table import *
        import matplotlib.pyplot as plt
        import matplotlib.pyplot as plt

In [2]: # starting with the western semi tone systems, lets generate an en
        interval_systems.get('semi_tones')

Out[2]: 1.0594630943592953

In [3]: # instantiate a C -1 class
        C_neg_1 = Hz(pitch_to_frequency.get('C')[-1])
        # make a semi_tone system with 128 frequencies
        semitone_sys_Cn1 = C_neg_1.make_system(system_type=interval_system
                                               system_size=range(0,128))
        semitone_sys_Cn1

Out[3]: {'freq_0': 8.18,
         'freq_1': 8.67,
         'freq_2': 9.18,
         'freq_3': 9.73,
```

M2_sys.ipynb hosted with ❤️ by GitHub view raw

notebook breaking down development of alternative musical systems

For those who are not familiar with the western musical system, the aim of this notebook is to present an exercise that depicts the mathematical process of taking a symbolic structure of discrete values, deconstruct it and nest itself, in its entirety, within the space of each discrete value. Embedding a mirroring substructure within the spaces of the superstructure.

Building a Wav

From here, let's use the frequencies from these systems to make a digital representation of their tones.

The below notebook briefly overviews the process of starting with a digit and constructing tones, then constructing systems based off those tones and analyzing a couple of the files to verify they are accurately rendered.

Using Systems to Construct Representations

```
In [3]: from wav_table import *
        from freq_table import *
```

```
In [2]: # use the Wav_Signal class to instantiate an object with a samplin
        A_4 = Wav_Signal(44100,
                         pitch_to_frequency.get('A')[4],
                         1.0)

In [3]: # construct a sine tone as a wav file
        A_4.make_simple_wav('sine')

In [4]: ls wav_files/

simple_440.0_sin.wav*

In [5]: # use the Hz class to instantiate an object of 440Hz
        A4 = Hz(pitch_to_frequency.get('A')[4])

In [8]: # construct semitone system from 440Hz
```

M3_sys.ipynb hosted with ❤️ by GitHub view raw

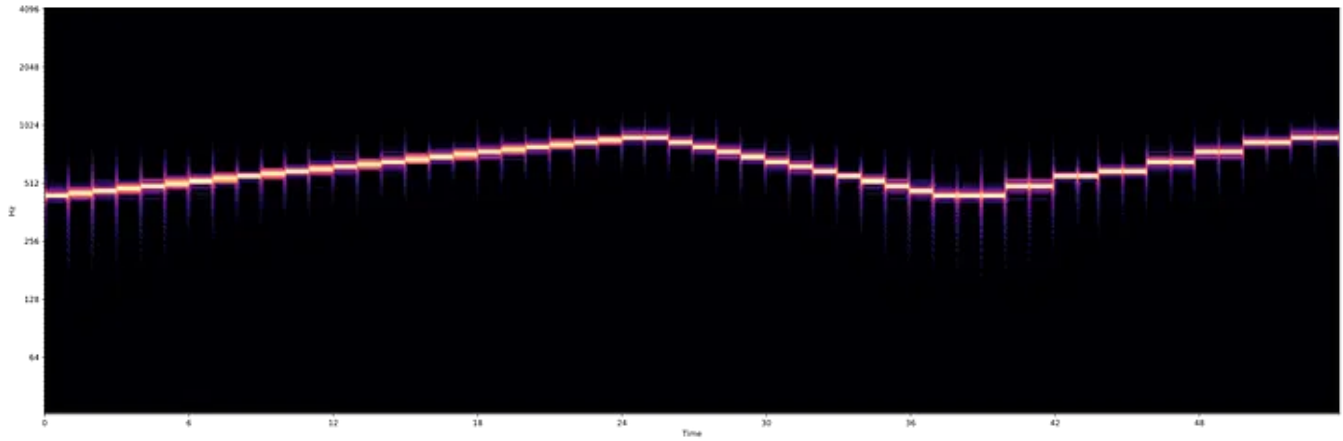
notebook surveying the construction and inspection of a digitally construction audio

Building Functional Music with Digital Representations of Tones

Figure 1 is three scales built from tones rendered from the above code.

1. chromatic quarter tone scale moving from 440Hz to 880Hz
2. chromatic semi tone scale moving from 880Hz to 440Hz
3. diatonic semi tone scale moving from 440Hz to 880Hz

audio fig 1

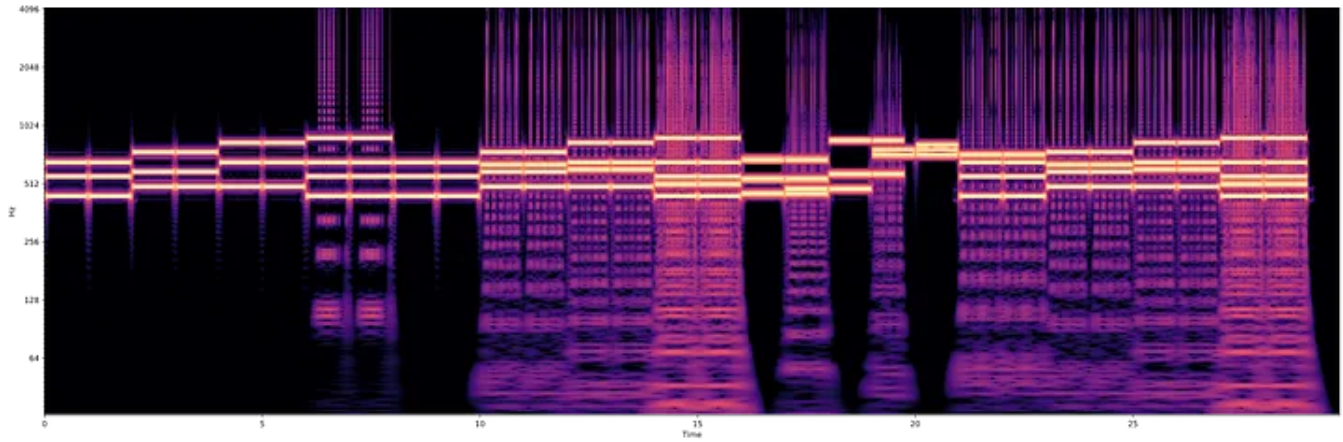


graph fig 1 | image generated by author's code

With the tones at our disposal, we could make *any* 'A' scale. I used these three just as a proof of concept. Along with scales, we can construct chords and progressions. Figure 2 is a progression built from the rendered tones.

1. The progression starts with a I ii V6 I
2. Then enters microtonal chords and ends with mixed chords utilizing tones from the diatonic and microtonal systems

audio fig 2



graph fig 2 | image generated by author's code

Novel Construction of Digital Audio

Below is a scan of an ink and pencil drawing I made some years ago. Below that is the same image converted in digital audio and displayed through a spectrograph. The data in each pixel is used to construct the frequency and amplitude for the audio information. In a future blog I can break down how this process works, but for now, I just wanted to show this to further elaborate that digital audio is a representation and not sound itself.

scan of an ink and pencil drawing by author | image by author

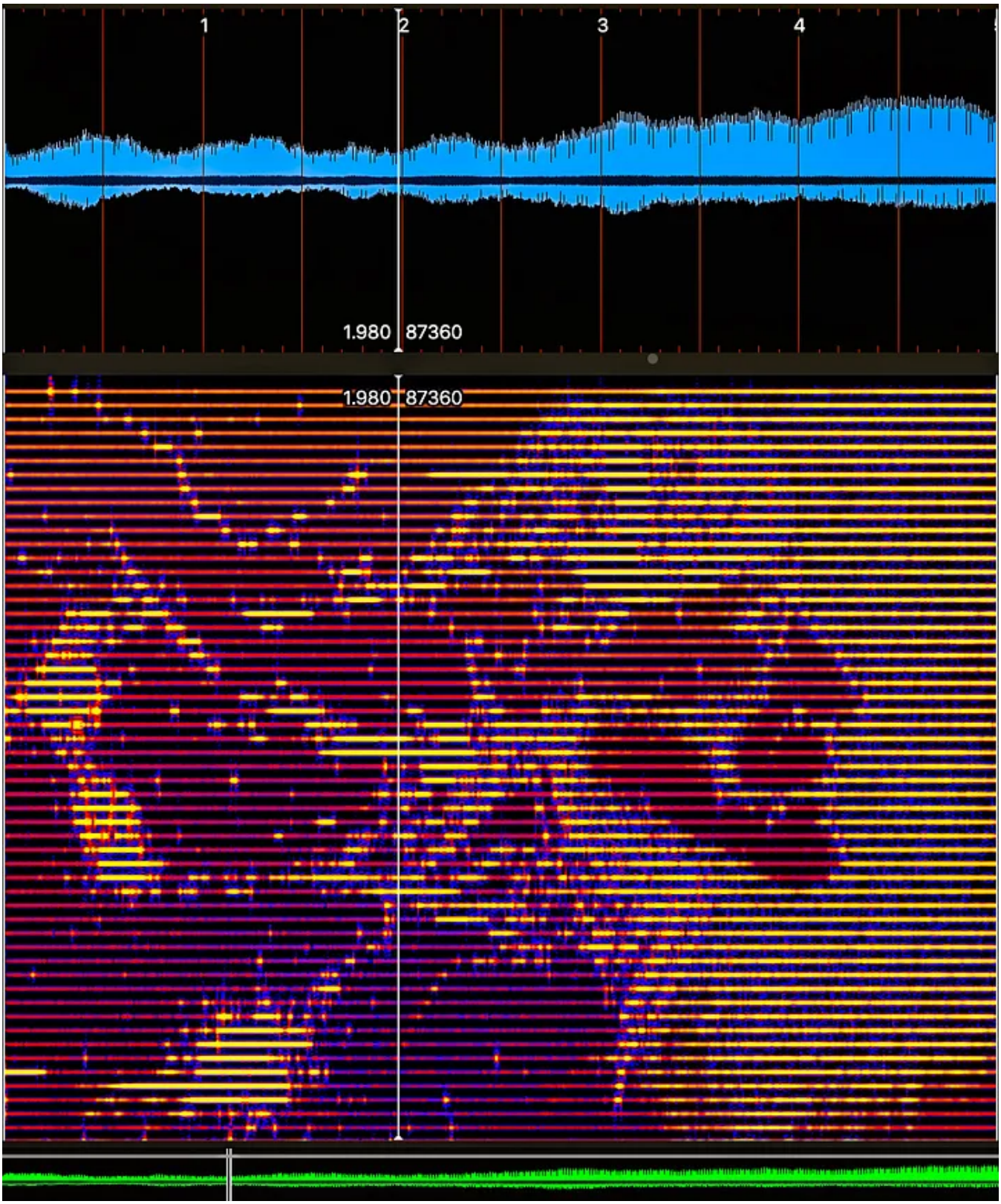


image rendered as a series of frequencies over time | image by author

ps. the actual sound it makes when played through speakers is unbearable

Manipulating Digital Audio

Developing digital renderings that mimic musical mathematical relations is one thing. It's nice to know the machine can instantiate the tones. But when a machine

is using purely mathematical precision, the perception of the sound is familiar but lacks qualities that we are expecting when hearing sound in the physical world.

Let's explore more of the module's functionality and use recorded audio. For this section, we will be using the soothing sounds of public domain mall muzak from the 1970s.

public domain music from the 70's

To manipulate the audio is similar to basic string and array manipulation in python.

```
#to know how many elements are in an given array of audio
num_of_ele = seconds*sample_rate
```

Each element is a sample containing a snapshot of audio data.

Below is a notebook that walks through a series of single manipulations on audio arrays and concludes in an operation of chaining the effects sequentially and passing the audio through. All resulting examples can be found beneath the notebook.

Manipulating Digital Audio

In [1]:
```
from wav_table import *
```

In [5]:
```
# load the elevator music
wav = '/Volumes/S200408/backup/S190813/Coding/wav_utilities/wav_ut
analyze_wav(wav)
```

```
sampling rate of file: 48000 Hz
bit depth of file: 32 bits
```

In [37]:

```
In [37]: cd manipulated_wav

/Volumes/S200408/backup/S190813/Coding/wav_utilities/wav_utilities/tools/manipulated_wav

In [7]: # split the evelator music into 30 second chuncks of audio
        split_single_wav_s(wav, 30)

wav file is cut in to equal parts with 30 second sections

In [8]: wav = 'cut_wav_0.wav'
```

M4_sys.ipynb hosted with ❤️ by GitHub view raw

notebook walking through a series of manipulations

Results of Audio Manipulation

Reversed Audio

There are a couple ways to reverse audio in python, but the main principle is to extract the audio data into an array and reverse the array then convert it back to an audio file. One simple way to do this:

```
sr, y = read(wav)
reverse_y = y[::-1]
write(outfile, sr, reverse_y)
```

Time Stretch Audio

Here are three examples of Time Stretch Algorithms that slow the audio. Depending how you factor the change in the sample rate, you can achieve different sounding stretches.

algorithm that multiples the sample rate by a factor greater than 1 to slow

algorithm that divides the sample rate by a factor less than 1 to slow

algorithm using scipy that broadcasts a multiplying factor across scipy's data output when reading in a wav

The fundamental idea is that you want to make the sample rate longer in order to stretch the musical content over longer intervals. To speed up the audio, you do the opposite.

Below are two examples of speeding up the audio.

algorithm that multiples the sample rate by a factor less than 1 to speed up

algorithm that divides the sample rate by a factor greater than 1 to speed up

Pitch Shift Audio

pitch shift up 100Hz

The key to finding a method of pitch shifting is to change the speed of the sample rate, without causing the wav to time stretch. To achieve this, I divided the amount of Hz I wanted to shift the pitch by the sample rate then separated the file into left and right channels. Once I did that, I used the *Fast Fourier Transform* on each sample in each channel and transposed the frequencies then reversed the process and stitched the two channels back together.

Delay Audio

delay set to 1000ms with a factor of 0.5 fade of repeats with 3 repeats

If you look at the spectrograph plotting the audio from this file, you can see how the delay blurs the middle of the file, but has distinct repetitions of three at the beginning and end where there is considerably less overlapping iterations of the

signal. In order to delay digital audio, you must manipulate the bytes by generating an offset. The logic to do so is below:

```
def delay(bytes,params,offset_ms):
    # generate offset
    offset= params.sampwidth*offset_ms*int(params.framerate/1000)

    # add silence in the beginning
    beginning= b'\0'*offset

    #remove space from the end
    end= bytes[:-offset]

    # concat bytes with both beginning and end with sample width
    return add(bytes, beginning+end, params.sampwidth)
```

Using Code Blocks like Chained Effects Pedals

After structuring algorithms in the sequence:

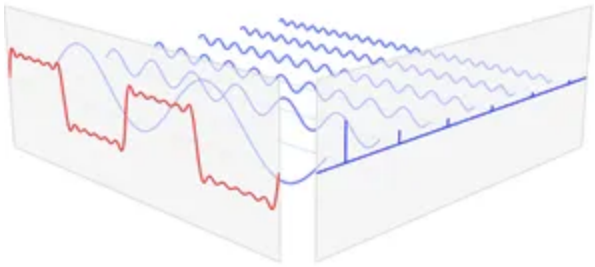
Reverse -> Delay -> Stretch -> Delay half Speed -> Layer Reverse

And passing the digital signal through, the below audio resulted.

sequential algorithms effects

Classifying Digital Audio

Everything up until this point has been meddling with the mathematical mechanisms that render digital audio. Essentially playing with the audio arrays. For the rest of this article, we will examine more in depth analysis of audio with feature extraction and how a machine can model audio using these features.



visualization of FFT | source

To extract these features, we will use methods found in Fourier Analysis; mainly: the Fast Fourier Transform. FFT is a method of taking a complex wave and parsing it into subsequent simpler waves. It allows us to access features based in the frequency domain. In the notebook below we will explore:

1. The Spectral Centroid
2. Spectral Bandwidth
3. Mel-Frequency Cepstral Coefficients
4. Chroma Features

To briefly explain these:

Spectral Centroid

The spectral centroid can be thought of as measuring the '*brightness*' of a given sound. It calculates the weighted mean of frequencies in a given sample.

$$\text{Centroid} = \frac{\sum_{n=0}^{N-1} f(n)x(n)}{\sum_{n=0}^{N-1} x(n)}$$

where x(n) represents the weighted frequency value, or magnitude, of bin number n, and f(n) represents the center frequency of that bin.

Spectral Bandwidth

The spectral bandwidth is related to the spectral '*resolution*' of a sound. It is defined as *the width of the band at one-half the peak maximum frequency per sample*.

Mel-Frequency Cepstral Coefficients

Built from a power cepstrum, the Mel-Frequency Cepstral is a representation of the short-term power spectrum of a sound. The coefficients are derived by examining a sample via FFT, then mapping the powers onto a mel scale. Then taking the log of each mel frequency. After that, performing as discrete cosine transform as if the list of log mel powers were an isolated signal. The resultant amplitudes of *that* 'signal' are the coefficients. The coefficients accurately describe the shape of the spectral

Chroma extractions show intensity along a vector of 0–11. This vector represents pitch class set and reveals the amount of each pitch class that is present. In other words, it is telling us what harmonic frequencies are present in the signal.

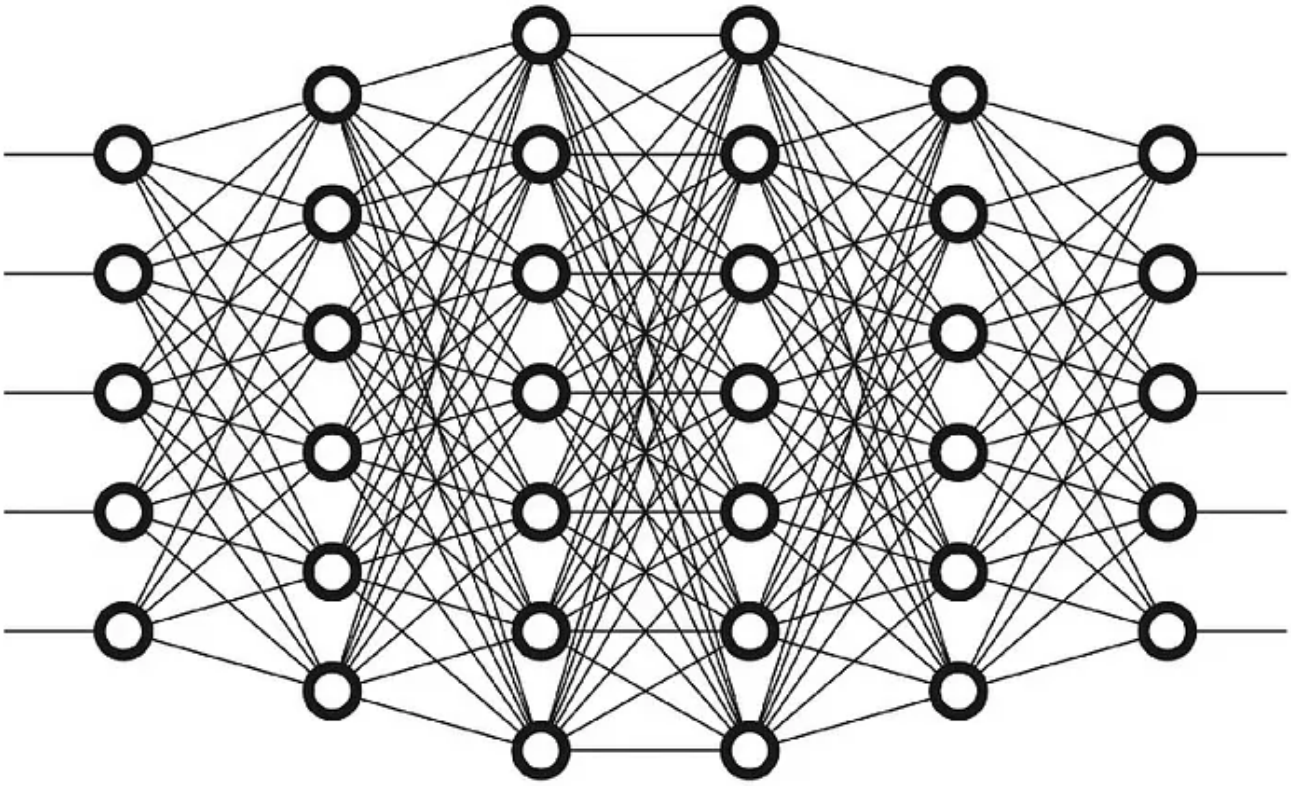
This is only a handful of features that are present in a signal, but with these and a few others shown in this notebook, we can begin modeling signals.

notebook showing examples of feature extraction for signal modeling

It's interesting, to model a digital audio signal is to extract representations of the characteristics of what is already a representation.

Modeling

Using the G. Tzanetakis collection, lets model music genre using these feature extractions. The collection is comprised of 1000 audio tracks each 30 seconds long. It contains 10 genres, each represented by 100 tracks. The tracks are all 22050 Hz monophonic 16-bit wav files. Once we have extracted all the data from the collection, let's start with our simple but good friend, the artificial deep neural network.



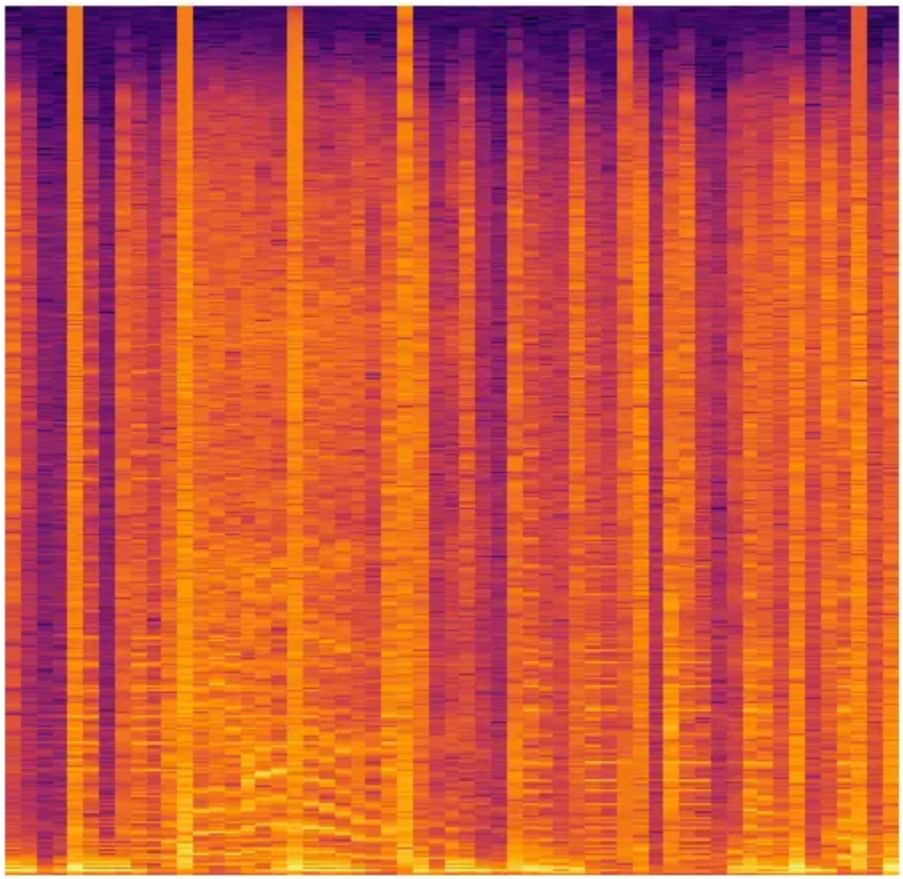
DeepANN Classifier | source

notebook building and implementing a simple DeepANN

The model achieves an accuracy of 69.50%.

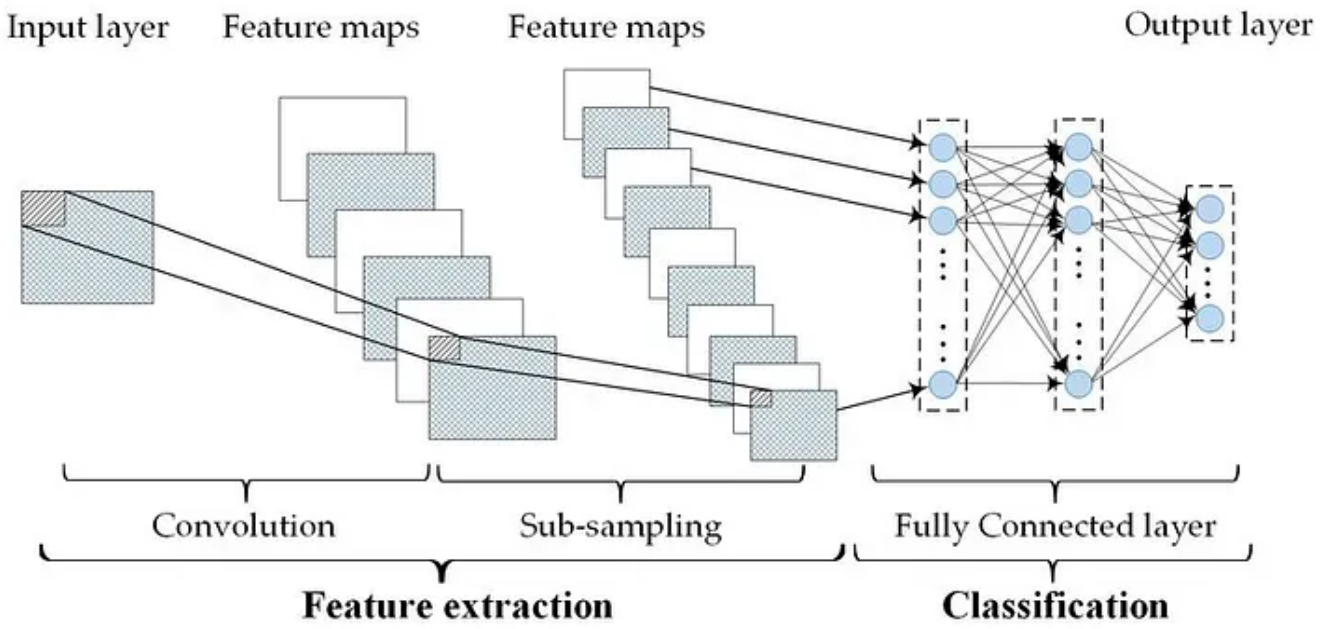
Considering that modeling is a method of engineering representations of the signals, let's look at other ways we could model the data. Rather than extracting features representing different values within a sample, let's use the image of the digital audio. To do this, we can take each wav file and generate a spectrograph. We can then use a convolutional neural network (CNN) on the spectrographs.

Here is an example of the spectrographs we will train the CNN on.



spectrograph of a 10 second wav file from gtzan collection, genre: hip hop | image generated by author's code

Rather than training on the extracted features themself to understand the musical profile of a genre, the CNN will learn the visual impression of the files to understand what characterizes a genre.

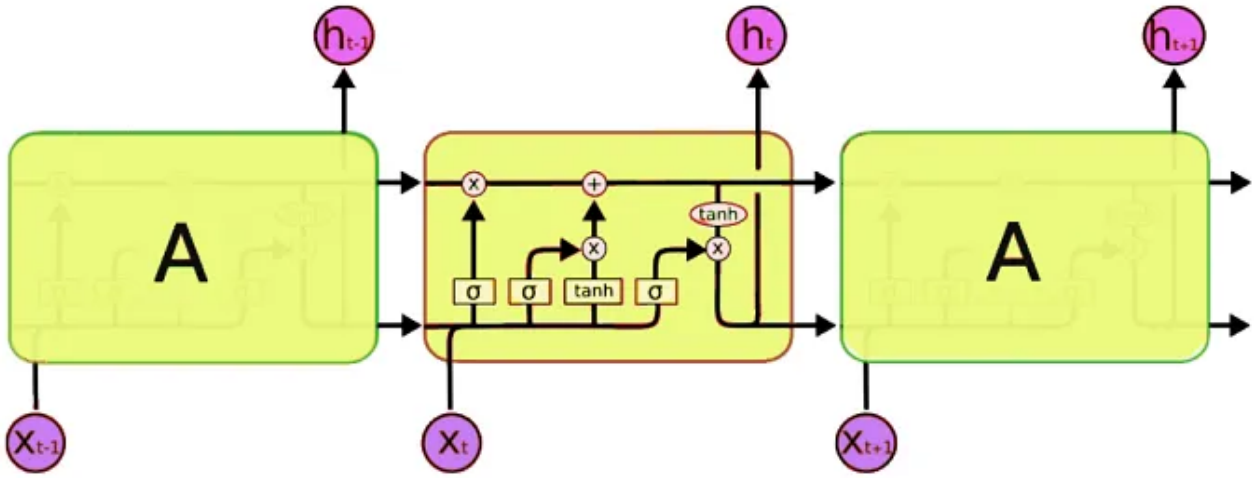


CNN classifier | source

notebook building and implementing a CNN

While the idea of using a spectrograph to represent the audio is interesting, using it to train a classifier doesn't seem to be the most effective method. Not to mention the training time on the CNN was much more expensive in memory and time.

Speaking of memory and time, let's revert back to using feature extraction from raw audio and try out a long short term memory (LSTM) model.



LSTM | source

notebook building and implementing LSTM

We reach an accuracy of 66.66% — a little less than 3% from the original neural net model we started this section with. Here the LSTM is effective, but I'd still go with the ANN for a classification problem. But let's see what else LSTMs can do.

Generate Digital Audio

In my article Generative AI: Malfunctions, Imperfections, Irrationality and Expression I discuss some opinions about if and how an AI can '*generate*'. Briefly, to overview, I hold the opinion that AI *can* generate, thus create, under certain circumstances. Before we attempt having an AI generate music, let's use the predict function to see how well it can replicate music.

In this notebook, I take three audio files.

1. *'Magic Spells' by Crystal Castles*
2. *'Two Can Win' by J Dilla*
3. *'Music in Twelve Parts, Part 3' by Philip Glass Ensemble (live performance in 1975)*

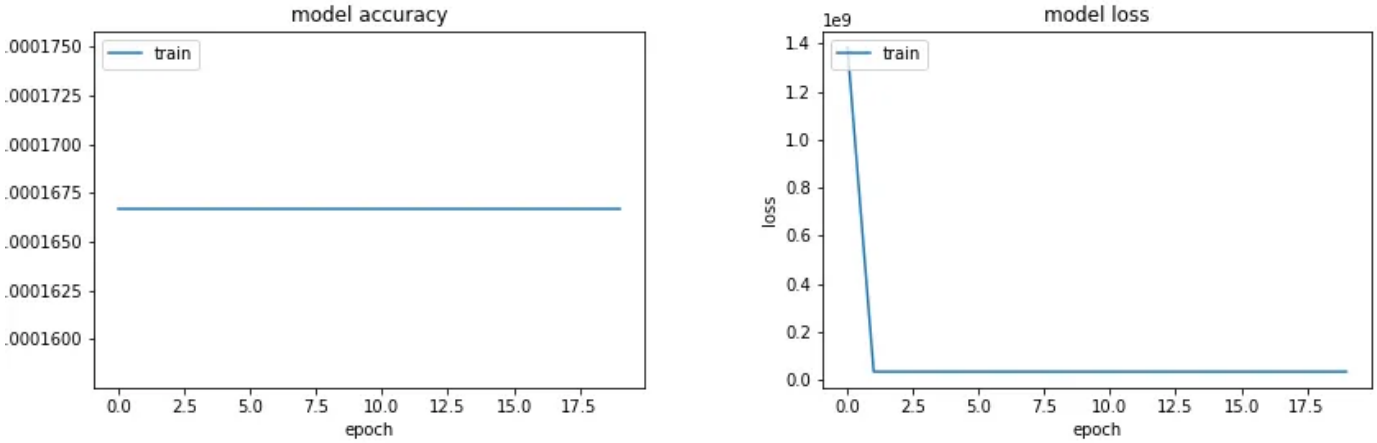
The task here is to use the higher dimension audio data, the array of amplitudes that we used in manipulating digital audio, to see if the LSTM can predict the proper sequence of amplitude per sample.

notebook building and implementing LSTM for audio generation

As we can see, the LSTM can be trained to do a fairly accurate job. Generated audio below:

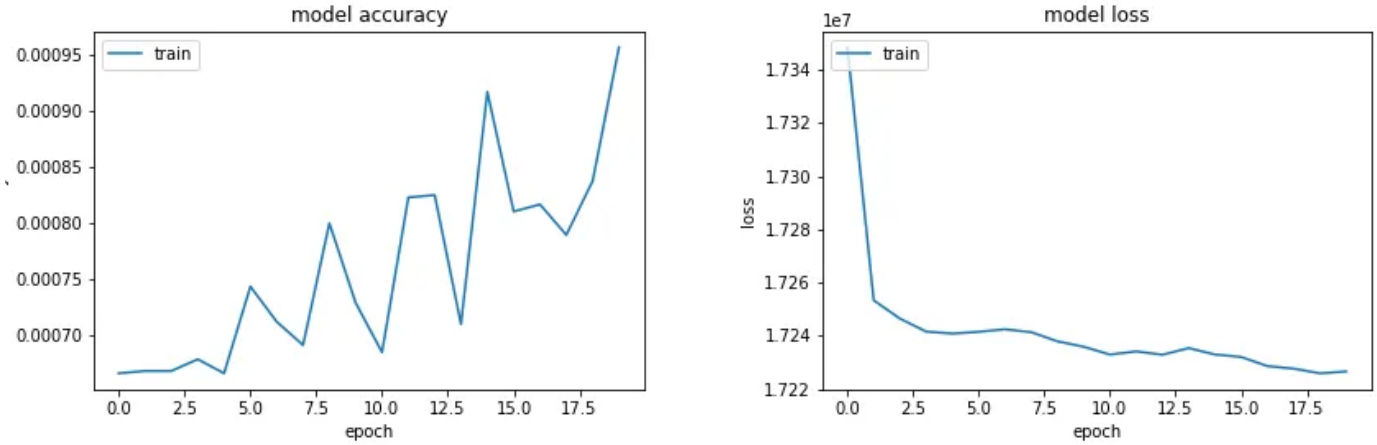
actual audio from three tracks spliced together

model under performing



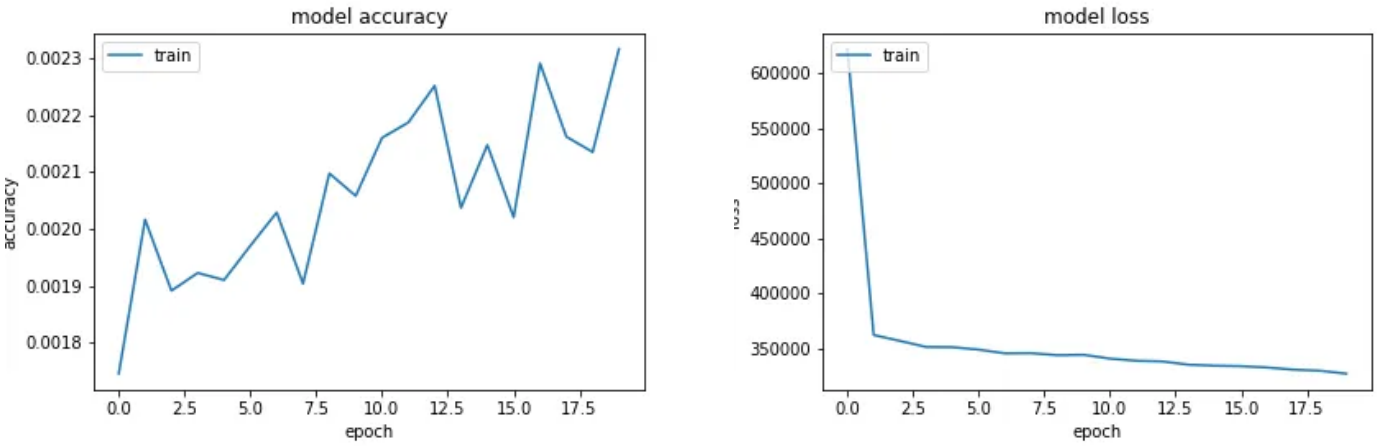
training performance of first model | images generated by author's code

model learning but not quite understanding volume



training performance of second model | images generated by author's code

model learns to predict amplitude accurately and recreates audio



training performance of third model | images generated by author's code

For this next experiment in generating audio, I won't be showing a notebook, because each model took several days if not longer to train (one model took close to a week with a GPU!). But for those who are interested to read the model's code, here is the final model's architecture. Depending on how I preprocessed the data, the model ending up producing some interesting results.

final LSTM architecture

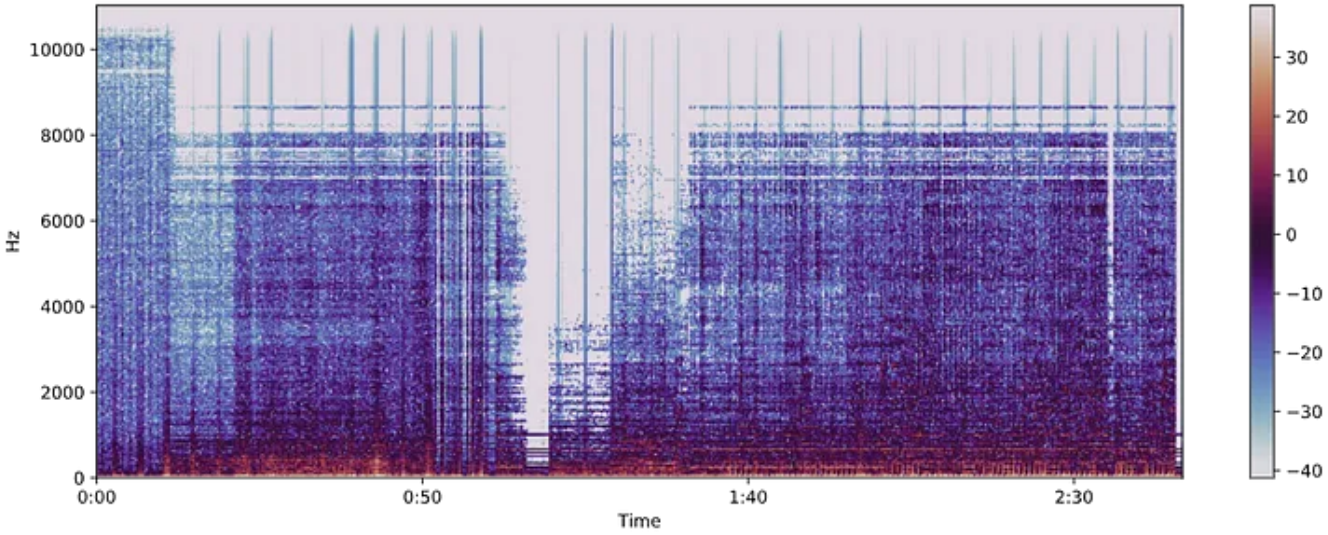
One of the main differences of this model, besides the numerous added layers, is the Time Distributed Dense Layering that everything is built within. This allows for the LSTM to consistently return full sequences for every step within and between layers while training. It also produces an output that is a full sequence, instead of a single sample at a time.

Rather than remain at a surface level of amplitude, in this experiment I wanted to train models on lower dimensional spectral features extracted via Fourier Analysis. And instead of training on a single audio file with a complex form, I trained on full albums and live performances of an artist. I continued to use Crystal Castles, J Dilla and Philip Glass Ensemble. Initially, I split entire albums into 10 second sections of audio, and eventually trained one model on entire tracks uncut.

During the training process, I saved various stages of the different model's developments and built a random sequence generator from each artist's extracted feature data. I then tasked each model from various stages of learning to generate audio to the best it could. Below are outputs from different stages strung together. The audio is not altered in any fashion and is placed in chronological order respective to the model's stages of learning.

**warning: it is a little noisy at beginning*

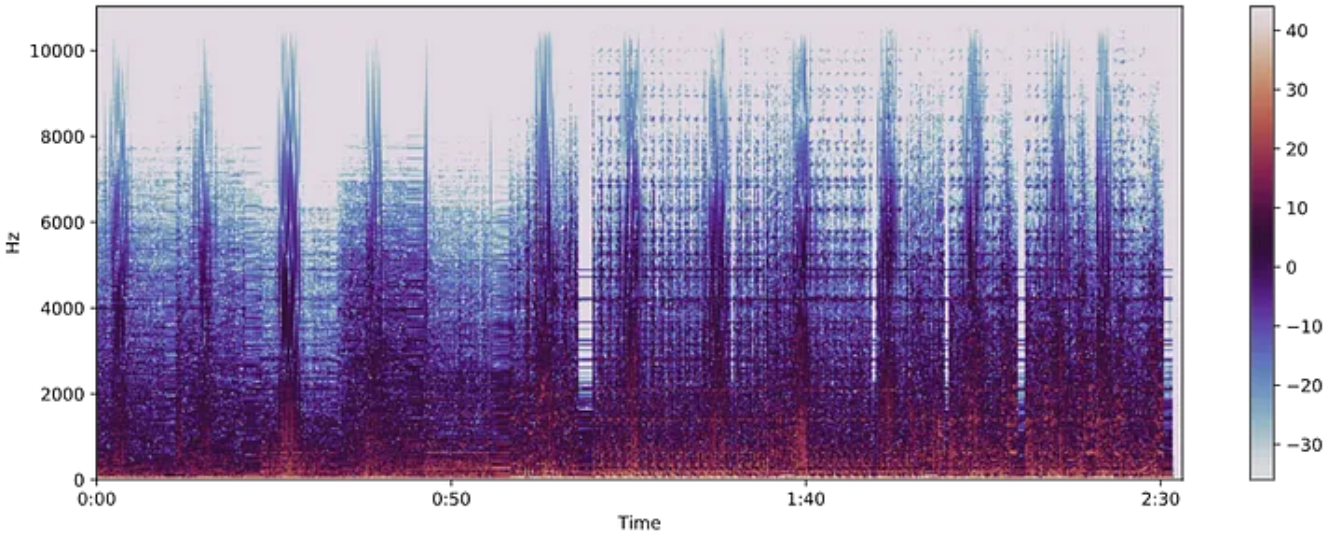
you might find this LSTM hanging out in bushwick near johnson ave & flushing, south of jefferson after quarantine



spectrograph of AI_LSTM_Generated_Crystal Castle | image generated by author's code

**there's something beautiful and imperfect occurring between :50s and 1:40s*

you might find this LSTM hanging out at Dally in the Ally in Woodbridge after quarantine



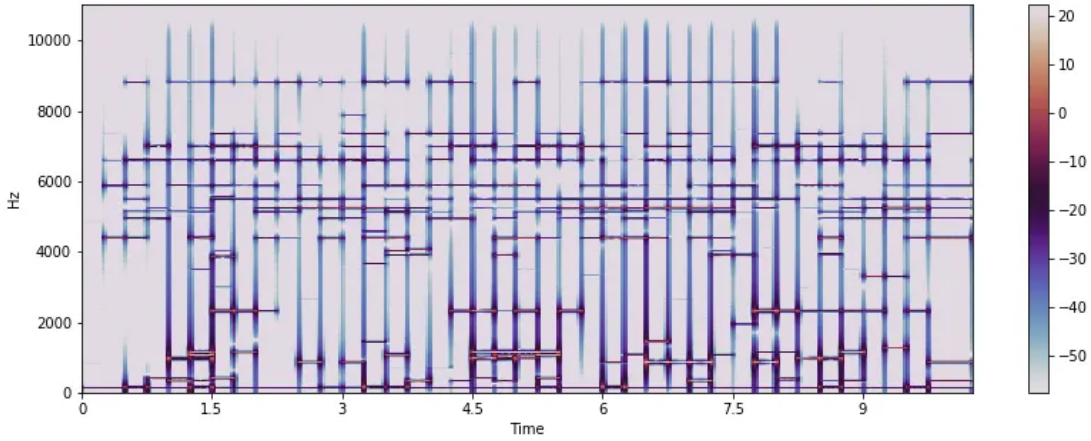
spectrograph of AI_LSTM_Generated_J Dilla | image generated by author's code

What really amazes me, is: what parts of audio the models learn first, given the data, and how quickly it goes from seemingly just tones and sounds to a defined understanding of rhythms. The output reminded me greatly of tuning through fm radio signals, looking for the right station, while driving upstate.

Now, in these two examples, there is some serious overfitting, as well some nuances. In fact, with the J Dilla track, it is interesting that the model kept spitting out loops. I wonder if that has anything to do with J Dilla's use of samples. Whereas in Crystal Castles' music, there is repetition, but it is people repeating words, rhythms and melodies while J Dilla tracks are verbatim sampled repetitions of records and tape. I also found it fascinating that in the J Dilla track, the human voice was clear and defined words could be heard — you can see the vocal overtones in the later half of the spectrograph; whereas in the Crystal Castles track, the voice was present but appears as a vague silhouette of words that blends seamlessly in with the instrumentation, rather than defined utterances.

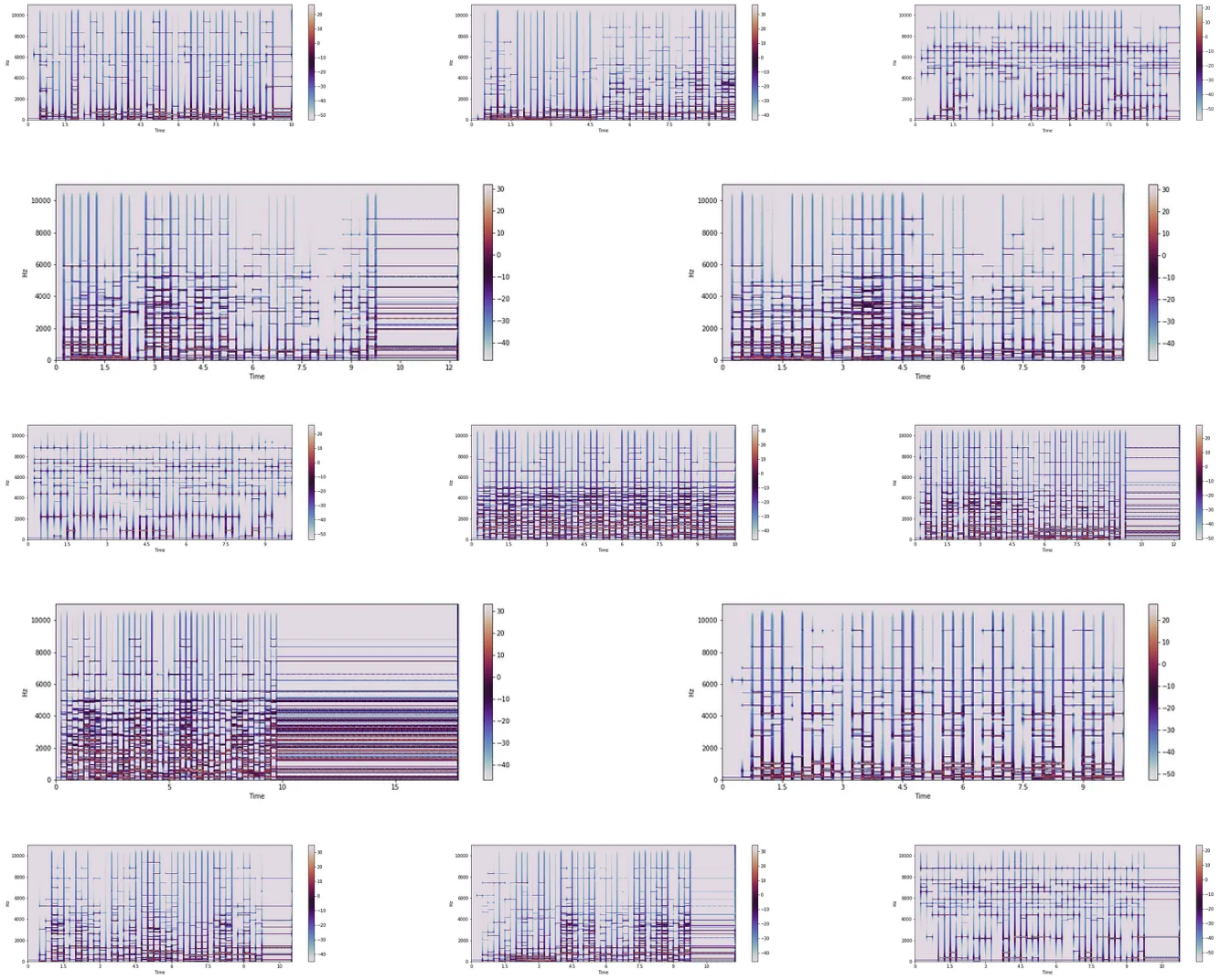
Lastly, let's listen to the Philip Glass Ensemble model. Something interesting is happening here. Rather than listening through the evolution from white noise to intelligent simulations, I want to zoom in and listen to the musical choices occurring here, as well as the timbre. In both Crystal Castles and J Dilla, each mature AI would oscillate between verbatim simulation and an edge of error that allowed for some glitchy imperfection that is reminiscent of a decision. With the Glass model, though, all the outputs are passages of music that cannot be exactly found in the live recording.

AI_LSTM_Philip Glass Ensemble



spectrograph of AI_LSTM_Glass_2_1_6 | image generated by author's code

Below are a handful of spectrographs that I find beautiful. The AI is understanding frequency and rhythm and generating signals that are not present in the actual recordings it was trained on. However, it is lacking in timbre and sounds like an analog synth; but to me, feels oddly genuine and has a ghostly influence of Philip Glass, but is something augmented, creative and unique.



all spectrographs generated by author's code

Below is a compilation of outputs I found interesting. The audio is unaltered.

compilation of GlassAI

Conclusion

> "And so art is everywhere, since artifice is at the very heart of reality. And so art is dead, not only because its critical transcendence is gone, but because reality itself, entirely impregnated by an aesthetic which is inseparable from its own structure, has been confused with its own image. Reality no longer has the time to take on the appearance of reality. It no longer even surpasses fiction: it captures every dream even before it takes on the appearance of a dream."

This, along with the concept of ***Hyperreality*** has been in the back of my mind throughout. And while I know Baudrillard is referring to a theoretical discussion of reality as a system of objects, something about this resonated with me when I began exploring DSP. I thought of digital audio as a representation, a simulation of sound. One that could not be distinguished from a naturally occurring sound. And technically, it is. But ending with the GlassAI, left me feeling that through simulation and the intentional development of replicant mediums, we may find a dimension of creativity that we simply could not reach without a technological bridge. Circling back to the idea that AI can be a creative and collaborative tool. I conclude with this notebook.

notebook sending GlassAI signal through sequential algorithm

output from GlassAI signal through sequential algorithms

Python Artificial Intelligence Data Science Music Philosophy

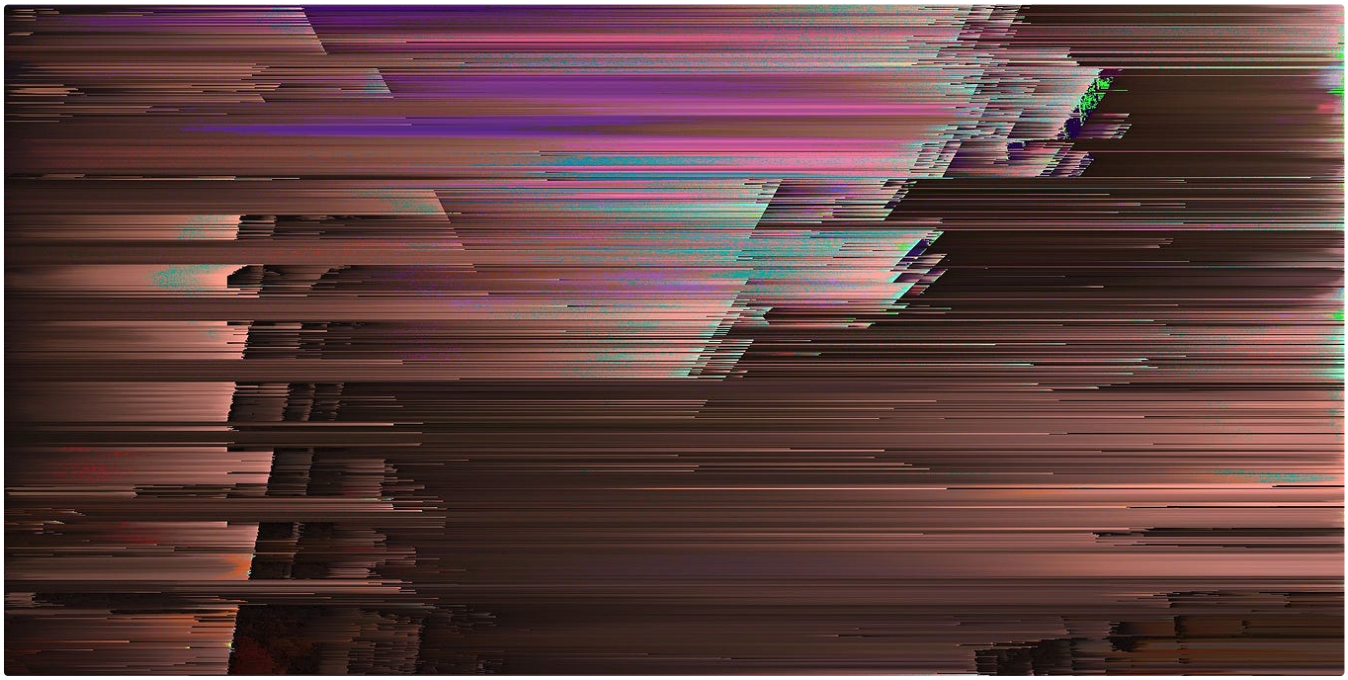
Follow

Written by Matthew R Finch

85 Followers · Writer for Towards Data Science

Data Scientist | Machine Learning Engineer | Creative and Critical Thinker

More from Matthew R Finch and Towards Data Science



Matthew R Finch

Sound Data

Visualization is the standard method of interpreting data. It is something we are all familiar with. However, it is not the only way to...

5 min read · Nov 12, 2019

162

Antonis Makropoulos in Towards Data Science

How to Build a Multi-GPU System for Deep Learning in 2023

This story provides a guide on how to build a multi-GPU system for deep learning and hopefully save you some research time and...

10 min read · Sep 17

537 11

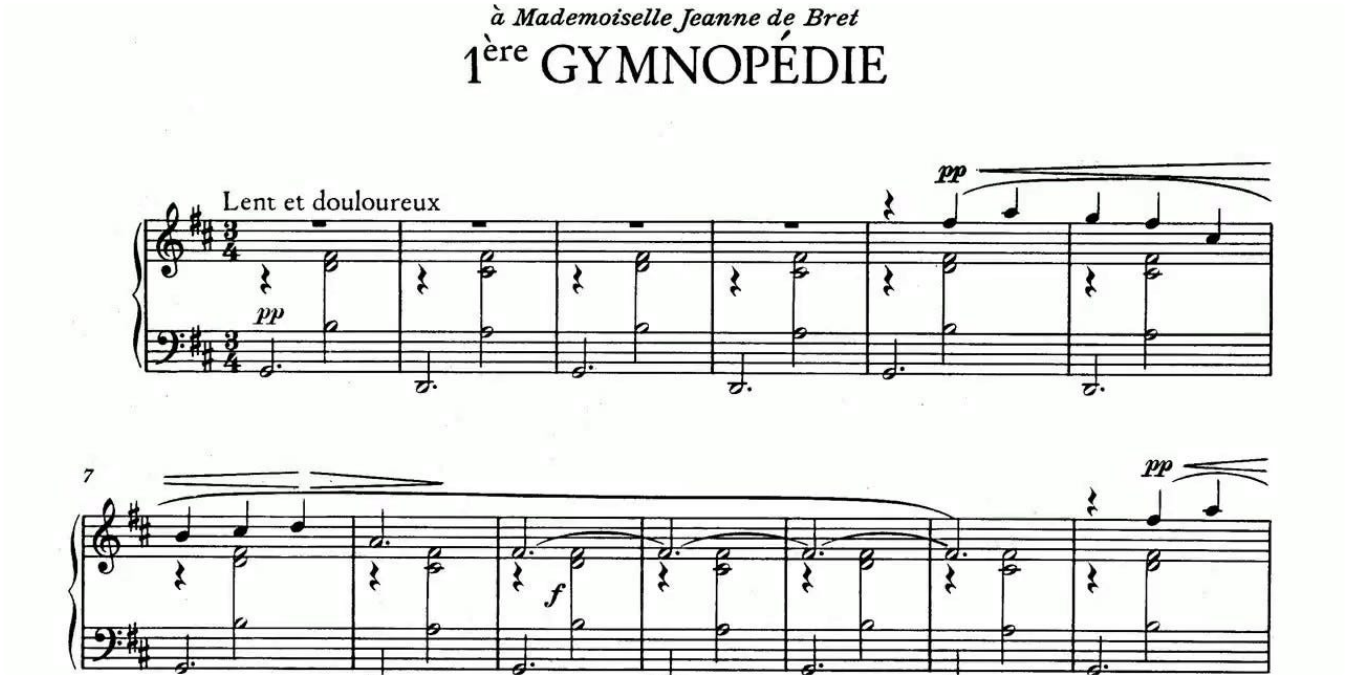
Khouloud El Alami in Towards Data Science

Don’t Start Your Data Science Journey Without These 5 Must-Do Steps From a Spotify Data Scientist

A complete guide to everything I wish I’d done before starting my Data Science journey, here’s to acing your first year with data

18 min read · 4 days ago

878 5



Matthew R Finch

Praise, Questions and Critique: Spotify API

Recently I built a Multiclass Music Genre Classification Model with Python using audio data from Spotify’s API. I was satisfied with the...

13 min read · Dec 18, 2019

554

See all from Matthew R Finch

See all from Towards Data Science

Recommended from Medium

Technocrat in CoderHack.com

Introduction to LibROSA

LibROSA is a Python package for audio and music analysis. It provides various functions to quickly extract key audio features and metrics...

3 min read · Sep 15

Daniel in d*classified

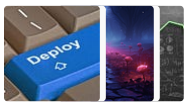
Exploring the Potential of AI for Music Emotion Recognition and Generation

Through analyzing instrumental audio content, Lim Xi Chen Terry has explored various models and techniques to classify audio clips based on...

8 min read · May 24

56

Lists



Predictive Modeling w/ Python
20 stories · 428 saves

ChatGPT
21 stories · 169 saves

Coding & Development
11 stories · 197 saves

ChatGPT prompts
24 stories · 434 saves

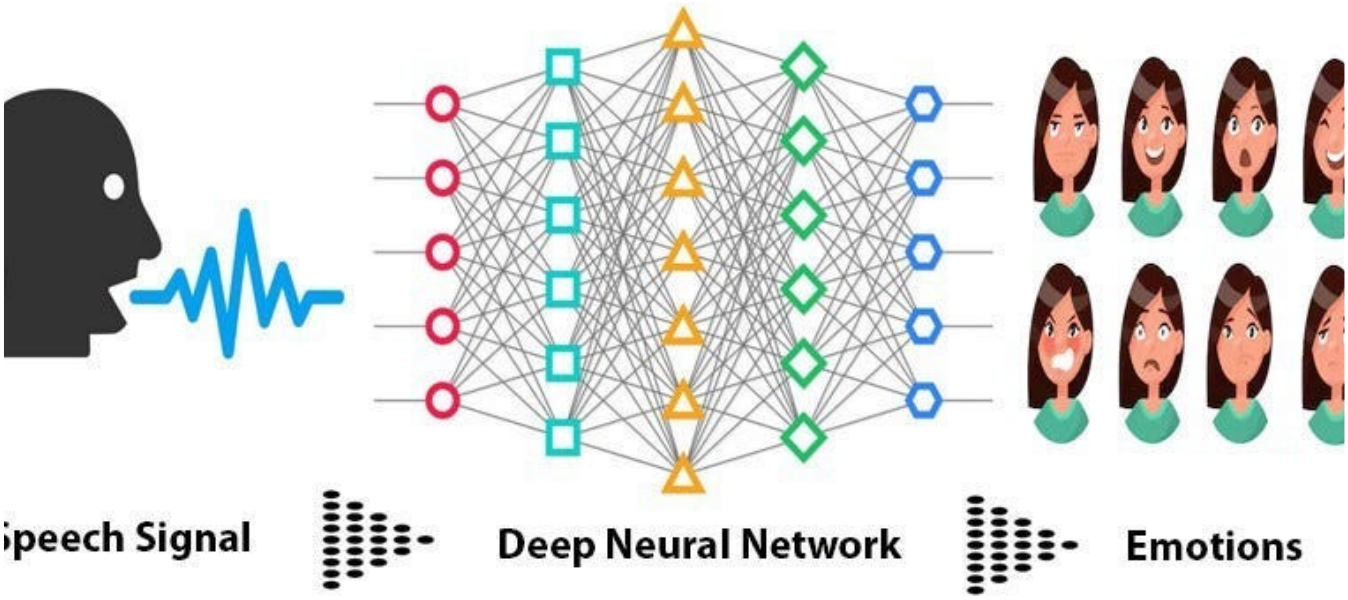
Bluetick Consultants

The Future of Automatic Speech Recognition—Whisper

Introduction

4 min read · Jun 2

1



Apoorv Pathak

Speech Emotion Recognition(Part 2)

5 min read · Apr 1

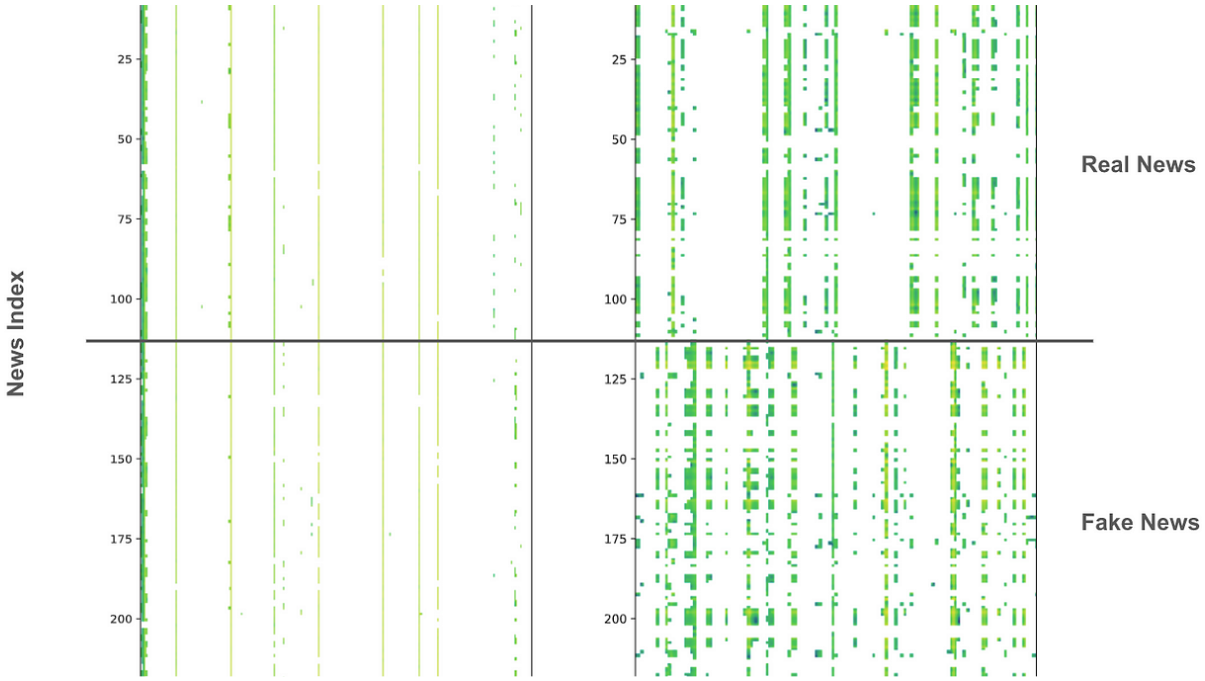
Kachkure Prathamesh

Text to Speech Synthesizer

Speech synthesis technologies are more in demand than ever right now. Speech synthesis is used by companies, movie studios, game...

8 min read · May 3

351 4



Sherry Wu in Stanford CS224W GraphML Tutorials

Spread No More: Twitter Fake News Detection with GNN

By Li Tian, Sherry Wu, Yifei Zheng as part of the Stanford CS224W course project.

11 min read · May 16

1

See more recommendations