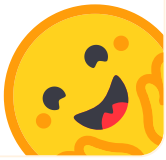


Introduction to audio data

By nature, a sound wave is a continuous signal, meaning it contains an infinite number of signal values in a given time. This poses problems for digital devices which expect finite arrays. To be processed, stored, and transmitted by digital devices, the continuous sound wave needs to be converted into a series of discrete values, known as a digital representation.

If you look at any audio dataset, you'll find digital files with sound excerpts, such as text narration or music. You may encounter different file formats such as `.wav` (Waveform Audio File), `.flac` (Free Lossless Audio Codec) and `.mp3` (MPEG-1 Audio Layer 3). These formats mainly differ in how they compress the digital representation of the audio signal.

Let's take a look at how we arrive from a continuous signal to this representation. The analog signal is first captured by a microphone, which converts the sound waves into an electrical signal. The electrical signal is then digitized by an Analog-to-Digital Converter to get the digital representation through sampling.

Sampling and sampling rate

Sampling is the process of measuring the value of a continuous signal at fixed time steps. The sampled waveform is *discrete*, since it contains a finite number of signal values at uniform intervals.

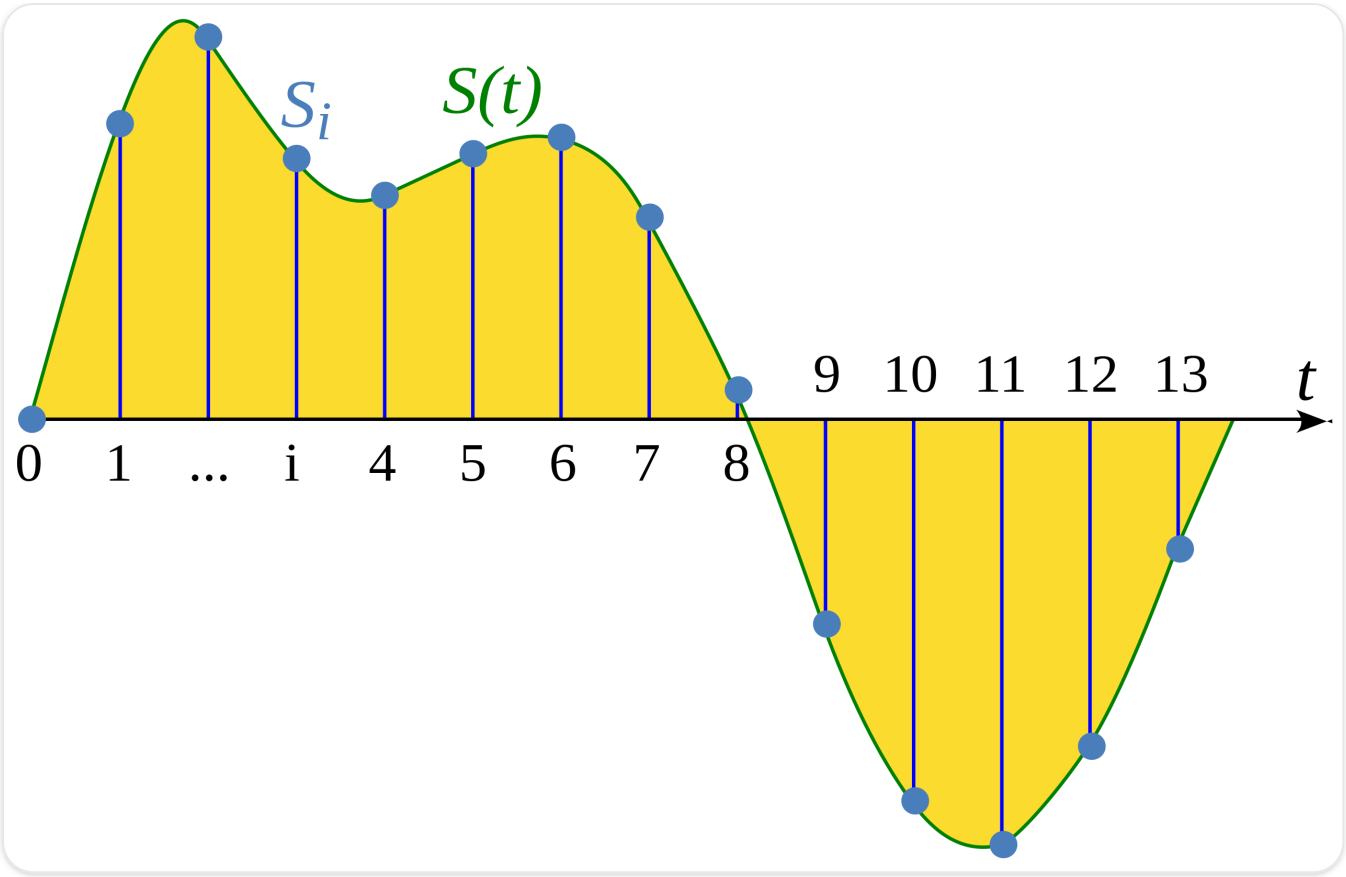


Illustration from Wikipedia article: Sampling (signal processing)

The **sampling rate** (also called sampling frequency) is the number of samples taken in one second and is measured in hertz (Hz). To give you a point of reference, CD-quality audio has a sampling rate of 44,100 Hz, meaning samples are taken 44,100 times per second. For comparison, high-resolution audio has a sampling rate of 192,000 Hz or 192 kHz. A common sampling rate used in training speech models is 16,000 Hz or 16 kHz.

The choice of sampling rate primarily determines the highest frequency that can be captured from the signal. This is also known as the Nyquist limit and is exactly half the sampling rate. The audible frequencies in human speech are below 8 kHz and therefore sampling speech at 16 kHz is sufficient. Using a higher sampling rate will not capture more information and merely leads to an increase in the computational cost of processing such files. On the other hand, sampling audio at too low a sampling rate will result in information loss. Speech sampled at 8 kHz will sound muffled, as the higher frequencies cannot be captured at this rate.

It's important to ensure that all audio examples in your dataset have the same sampling rate when working on any audio task. If you plan to use custom audio data to fine-tune a pre-trained model, the sampling rate of your data should match the sampling rate of the data the model was

pre-trained on. The sampling rate determines the time interval between successive audio samples, which impacts the temporal resolution of the audio data. Consider an example: a 5-second sound at a sampling rate of 16,000 Hz will be represented as a series of 80,000 values, while the same 5-second sound at a sampling rate of 8,000 Hz will be represented as a series of 40,000 values. Transformer models that solve audio tasks treat examples as sequences and rely on attention mechanisms to learn audio or multimodal representation. Since sequences are different for audio examples at different sampling rates, it will be challenging for models to generalize between sampling rates. **Resampling** is the process of making the sampling rates match, and is part of preprocessing the audio data.

Amplitude and bit depth

While the sampling rate tells you how often the samples are taken, what exactly are the values in each sample?

Sound is made by changes in air pressure at frequencies that are audible to humans. The **amplitude** of a sound describes the sound pressure level at any given instant and is measured in decibels (dB). We perceive the amplitude as loudness. To give you an example, a normal speaking voice is under 60 dB, and a rock concert can be at around 125 dB, pushing the limits of human hearing.

In digital audio, each audio sample records the amplitude of the audio wave at a point in time. The **bit depth** of the sample determines with how much precision this amplitude value can be described. The higher the bit depth, the more faithfully the digital representation approximates the original continuous sound wave.

The most common audio bit depths are 16-bit and 24-bit. Each is a binary term, representing the number of possible steps to which the amplitude value can be quantized when it's converted from continuous to discrete: 65,536 steps for 16-bit audio, a whopping 16,777,216 steps for 24-bit audio. Because quantizing involves rounding off the continuous value to a discrete value, the sampling process introduces noise. The higher the bit depth, the smaller this quantization noise. In practice, the quantization noise of 16-bit audio is already small enough to be inaudible, and using higher bit depths is generally not necessary.

You may also come across 32-bit audio. This stores the samples as floating-point values, whereas 16-bit and 24-bit audio use integer samples. The precision of a 32-bit floating-point

value is 24 bits, giving it the same bit depth as 24-bit audio. Floating-point audio samples are expected to lie within the [-1.0, 1.0] range. Since machine learning models naturally work on floating-point data, the audio must first be converted into floating-point format before it can be used to train the model. We'll see how to do this in the next section on Preprocessing.

Just as with continuous audio signals, the amplitude of digital audio is typically expressed in decibels (dB). Since human hearing is logarithmic in nature — our ears are more sensitive to small fluctuations in quiet sounds than in loud sounds — the loudness of a sound is easier to interpret if the amplitudes are in decibels, which are also logarithmic. The decibel scale for real-world audio starts at 0 dB, which represents the quietest possible sound humans can hear, and louder sounds have larger values. However, for digital audio signals, 0 dB is the loudest possible amplitude, while all other amplitudes are negative. As a quick rule of thumb: every -6 dB is a halving of the amplitude, and anything below -60 dB is generally inaudible unless you really crank up the volume.

Audio as a waveform

You may have seen sounds visualized as a **waveform**, which plots the sample values over time and illustrates the changes in the sound's amplitude. This is also known as the *time domain* representation of sound.

This type of visualization is useful for identifying specific features of the audio signal such as the timing of individual sound events, the overall loudness of the signal, and any irregularities or noise present in the audio.

To plot the waveform for an audio signal, we can use a Python library called `librosa`:

```
pip install librosa
```

Let's take an example sound called "trumpet" that comes with the library:

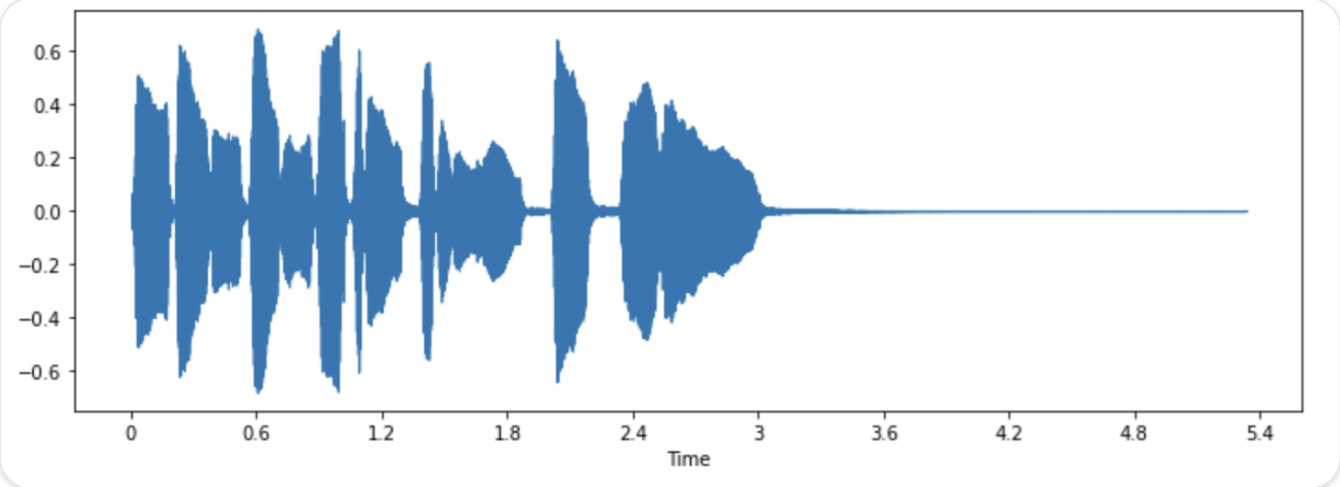
```
import librosa

array, sampling_rate = librosa.load(librosa.ex("trumpet"))
```

The example is loaded as a tuple of audio time series (here we call it `array`), and sampling rate (`sampling_rate`). Let's take a look at this sound's waveform by using librosa's `waveshow()` function:

```
import matplotlib.pyplot as plt
import librosa.display

plt.figure().set_figwidth(12)
librosa.display.waveshow(array, sr=sampling_rate)
```



This plots the amplitude of the signal on the y-axis and time along the x-axis. In other words, each point corresponds to a single sample value that was taken when this sound was sampled. Also note that librosa returns the audio as floating-point values already, and that the amplitude values are indeed within the [-1.0, 1.0] range.

Visualizing the audio along with listening to it can be a useful tool for understanding the data you are working with. You can see the shape of the signal, observe patterns, learn to spot noise or distortion. If you preprocess data in some ways, such as normalization, resampling, or filtering, you can visually confirm that preprocessing steps have been applied as expected. After training a model, you can also visualize samples where errors occur (e.g. in audio classification task) to debug the issue.

The frequency spectrum

Another way to visualize audio data is to plot the **frequency spectrum** of an audio signal, also known as the *frequency domain* representation. The spectrum is computed using the discrete Fourier transform or DFT. It describes the individual frequencies that make up the signal and how strong they are.

Let's plot the frequency spectrum for the same trumpet sound by taking the DFT using numpy's `rfft()` function. While it is possible to plot the spectrum of the entire sound, it's more useful to look at a small region instead. Here we'll take the DFT over the first 4096 samples, which is roughly the length of the first note being played:

```
import numpy as np

dft_input = array[:4096]

# calculate the DFT
window = np.hanning(len(dft_input))
windowed_input = dft_input * window
dft = np.fft.rfft(windowed_input)

# get the amplitude spectrum in decibels
amplitude = np.abs(dft)
amplitude_db = librosa.amplitude_to_db(amplitude, ref=np.max)

# get the frequency bins
frequency = librosa.fft_frequencies(sr=sampling_rate, n_fft=len(dft_input))

plt.figure().set_figwidth(12)
plt.plot(frequency, amplitude_db)
plt.xlabel("Frequency (Hz)")
plt.ylabel("Amplitude (dB)")
plt.xscale("log")
```

This plots the strength of the various frequency components that are present in this audio segment. The frequency values are on the x-axis, usually plotted on a logarithmic scale, while their amplitudes are on the y-axis.

The frequency spectrum that we plotted shows several peaks. These peaks correspond to the harmonics of the note that's being played, with the higher harmonics being quieter. Since the first peak is at around 620 Hz, this is the frequency spectrum of an E♭ note.

The output of the DFT is an array of complex numbers, made up of real and imaginary components. Taking the magnitude with `np.abs(dft)` extracts the amplitude information from the spectrogram. The angle between the real and imaginary components provides the so-called phase spectrum, but this is often discarded in machine learning applications.

You used `librosa.amplitude_to_db()` to convert the amplitude values to the decibel scale, making it easier to see the finer details in the spectrum. Sometimes people use the **power spectrum**, which measures energy rather than amplitude; this is simply a spectrum with the amplitude values squared.

> 💡 In practice, people use the term FFT interchangeably with DFT, as the FFT or Fast Fourier Transform is the only efficient way to calculate the DFT on a computer.

The frequency spectrum of an audio signal contains the exact same information as its waveform — they are simply two different ways of looking at the same data (here, the first 4096 samples from the trumpet sound). Where the waveform plots the amplitude of the audio signal over time, the spectrum visualizes the amplitudes of the individual frequencies at a fixed point in time.

Spectrogram

What if we want to see how the frequencies in an audio signal change? The trumpet plays several notes and they all have different frequencies. The problem is that the spectrum only shows a frozen snapshot of the frequencies at a given instant. The solution is to take multiple

DFTs, each covering only a small slice of time, and stack the resulting spectra together into a **spectrogram**.

A spectrogram plots the frequency content of an audio signal as it changes over time. It allows you to see time, frequency, and amplitude all on one graph. The algorithm that performs this computation is the STFT or Short Time Fourier Transform.

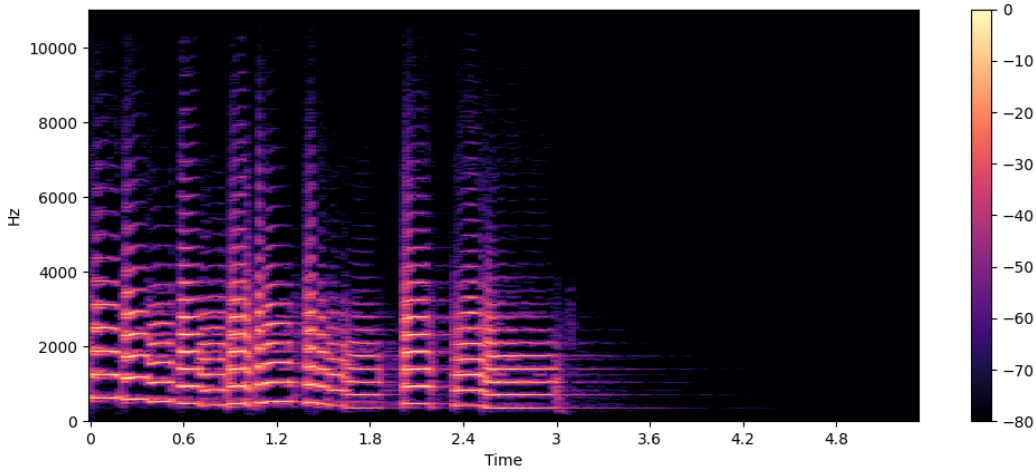
The spectrogram is one of the most informative audio tools available to you. For example, when working with a music recording, you can see the various instruments and vocal tracks and how they contribute to the overall sound. In speech, you can identify different vowel sounds as each vowel is characterized by particular frequencies.

Let's plot a spectrogram for the same trumpet sound, using librosa's `stft()` and `specshow()` functions:

```
import numpy as np

D = librosa.stft(array)
S_db = librosa.amplitude_to_db(np.abs(D), ref=np.max)

plt.figure().set_figwidth(12)
librosa.display.specshow(S_db, x_axis="time", y_axis="hz")
plt.colorbar()
```



In this plot, the x-axis represents time as in the waveform visualization but now the y-axis represents frequency in Hz. The intensity of the color gives the amplitude or power of the frequency component at each point in time, measured in decibels (dB).

The spectrogram is created by taking short segments of the audio signal, typically lasting a few milliseconds, and calculating the discrete Fourier transform of each segment to obtain its frequency spectrum. The resulting spectra are then stacked together on the time axis to create the spectrogram. Each vertical slice in this image corresponds to a single frequency spectrum, seen from the top. By default, `librosa.stft()` splits the audio signal into segments of 2048 samples, which gives a good trade-off between frequency resolution and time resolution.

Since the spectrogram and the waveform are different views of the same data, it's possible to turn the spectrogram back into the original waveform using the inverse STFT. However, this requires the phase information in addition to the amplitude information. If the spectrogram was generated by a machine learning model, it typically only outputs the amplitudes. In that case, we can use a phase reconstruction algorithm such as the classic Griffin-Lim algorithm, or using a neural network called a vocoder, to reconstruct a waveform from the spectrogram.

Spectrograms aren't just used for visualization. Many machine learning models will take spectrograms as input — as opposed to waveforms — and produce spectrograms as output.

Now that we know what a spectrogram is and how it's made, let's take a look at a variant of it widely used for speech processing: the mel spectrogram.

Mel spectrogram

A mel spectrogram is a variation of the spectrogram that is commonly used in speech processing and machine learning tasks. It is similar to a spectrogram in that it shows the frequency content of an audio signal over time, but on a different frequency axis.

In a standard spectrogram, the frequency axis is linear and is measured in hertz (Hz). However, the human auditory system is more sensitive to changes in lower frequencies than higher frequencies, and this sensitivity decreases logarithmically as frequency increases. The mel scale is a perceptual scale that approximates the non-linear frequency response of the human ear.

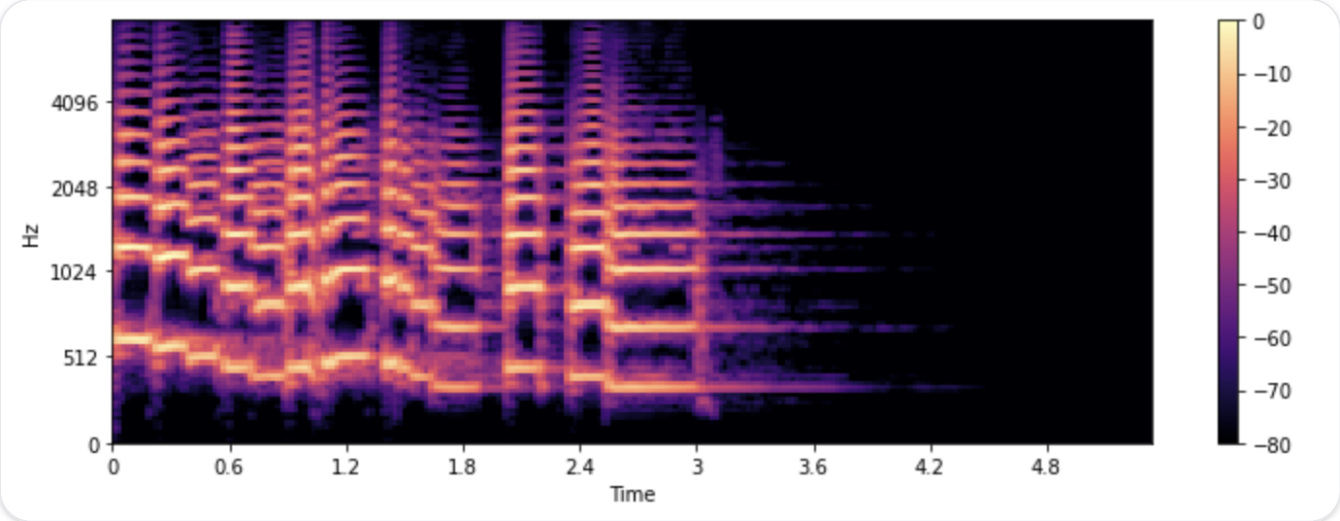
To create a mel spectrogram, the STFT is used just like before, splitting the audio into short segments to obtain a sequence of frequency spectra. Additionally, each spectrum is sent

through a set of filters, the so-called mel filterbank, to transform the frequencies to the mel scale.

Let's see how we can plot a mel spectrogram using librosa's `melspectrogram()` function, which performs all of those steps for us:

```
S = librosa.feature.melspectrogram(y=array, sr=sampling_rate, n_mels=128, fmax=8000)
S_dB = librosa.power_to_db(S, ref=np.max)

plt.figure().set_figwidth(12)
librosa.display.specshow(S_dB, x_axis="time", y_axis="mel", sr=sampling_rate, fmax=800
plt.colorbar()
```



In the example above, `n_mels` stands for the number of mel bands to generate. The mel bands define a set of frequency ranges that divide the spectrum into perceptually meaningful components, using a set of filters whose shape and spacing are chosen to mimic the way the human ear responds to different frequencies. Common values for `n_mels` are 40 or 80. `fmax` indicates the highest frequency (in Hz) we care about.

Just as with a regular spectrogram, it's common practice to express the strength of the mel frequency components in decibels. This is commonly referred to as a **log-mel spectrogram**, because the conversion to decibels involves a logarithmic operation. The above example used `librosa.power_to_db()` as `librosa.feature.melspectrogram()` creates a power spectrogram.

> 💡 Not all mel spectrograms are the same! There are two different mel scales in common use ("htk" and "slaney"), and instead of the power spectrogram the amplitude spectrogram may be used. The conversion to a log-mel spectrogram doesn't always compute true decibels but may simply take the `log`. Therefore, if a machine learning model expects a mel spectrogram as input, double check to make sure you're computing it the same way.

Creating a mel spectrogram is a lossy operation as it involves filtering the signal. Converting a mel spectrogram back into a waveform is more difficult than doing this for a regular spectrogram, as it requires estimating the frequencies that were thrown away. This is why machine learning models such as HiFiGAN vocoder are needed to produce a waveform from a mel spectrogram.

Compared to a standard spectrogram, a mel spectrogram can capture more meaningful features of the audio signal for human perception, making it a popular choice in tasks such as speech recognition, speaker identification, and music genre classification.

Now that you know how to visualize audio data examples, go ahead and try to see what your favorite sounds look like. :)

← What you'll learn Load and explore an audio dataset →