```
pip install librosa
```

```
Requirement already satisfied: librosa in /usr/local/lib/python3.10/dist-packages (0.10.2.post1)
Requirement already satisfied: audioread>=2.1.9 in /usr/local/lib/python3.10/dist-packages (from librosa) (3.0.1)
Requirement already satisfied: numpy!=1.22.0,!=1.22.1,!=1.22.2,>=1.20.3 in /usr/local/lib/python3.10/dist-packages (from librosa) (1.26.4)
Requirement already satisfied: scipy>=1.2.0 in /usr/local/lib/python3.10/dist-packages (from librosa) (1.13.1)
Requirement already satisfied: scikit-learn>=0.20.0 in /usr/local/lib/python3.10/dist-packages (from librosa) (1.3.2)
Requirement already satisfied: joblib>=0.14 in /usr/local/lib/python3.10/dist-packages (from librosa) (1.4.2)
Requirement already satisfied: decorator>=4.3.0 in /usr/local/lib/python3.10/dist-packages (from librosa) (4.4.2)
Requirement already satisfied: numba>=0.51.0 in /usr/local/lib/python3.10/dist-packages (from librosa) (0.60.0)
Requirement already satisfied: soundfile>=0.12.1 in /usr/local/lib/python3.10/dist-packages (from librosa) (0.12.1)
Requirement already satisfied: pooch>=1.1 in /usr/local/lib/python3.10/dist-packages (from librosa) (1.8.2)
Requirement already satisfied: soxr>=0.3.2 in /usr/local/lib/python3.10/dist-packages (from librosa) (0.4.0)
Requirement already satisfied: typing-extensions>=4.1.1 in /usr/local/lib/python3.10/dist-packages (from librosa) (4.12.2)
Requirement already satisfied: lazy-loader>=0.1 in /usr/local/lib/python3.10/dist-packages (from librosa) (0.4)
Requirement already satisfied: msgpack>=1.0 in /usr/local/lib/python3.10/dist-packages (from librosa) (1.0.8)
Requirement already satisfied: packaging in /usr/local/lib/python3.10/dist-packages (from lazy-loader>=0.1->librosa) (24.1)
Requirement already satisfied: llvmlite<0.44,>=0.43.0dev0 in /usr/local/lib/python3.10/dist-packages (from numba>=0.51.0->librosa) (0.43.0)
Requirement already satisfied: platformdirs>=2.5.0 in /usr/local/lib/python3.10/dist-packages (from pooch>=1.1->librosa) (4.2.2)
Requirement already satisfied: requests>=2.19.0 in /usr/local/lib/python3.10/dist-packages (from pooch>=1.1->librosa) (2.31.0)
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.10/dist-packages (from scikit-learn>=0.20.0->librosa) (3.5.0)
Requirement already satisfied: cffi>=1.0 in /usr/local/lib/python3.10/dist-packages (from soundfile>=0.12.1->librosa) (1.16.0)
Requirement already satisfied: pycparser in /usr/local/lib/python3.10/dist-packages (from cffi>=1.0->soundfile>=0.12.1->librosa) (2.22)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from requests>=2.19.0->pooch>=1.1->librosa) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests>=2.19.0->pooch>=1.1->librosa) (3.7)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from requests>=2.19.0->pooch>=1.1->librosa) (2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from requests>=2.19.0->pooch>=1.1->librosa) (2024.7.4)
```

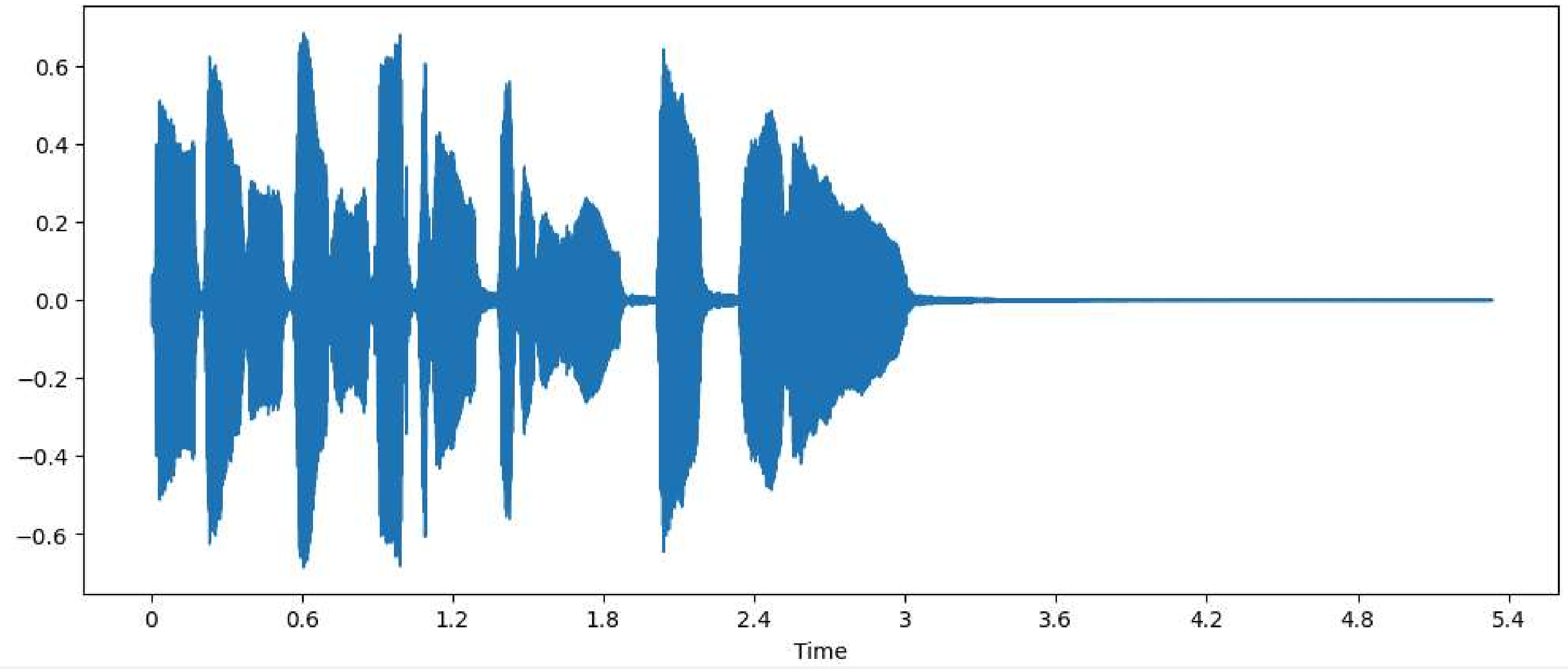
```
import librosa

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

```
Downloading file 'sorohanro_-_solo-trumpet-06.ogg' from 'https://librosa.org/data/audio/sorohanro_-_solo-trumpet-06.ogg' to '/root/.cache/librosa'.
```

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)
```

```
<librosa.display.AdaptiveWaveplot at 0x7f81f08d6ec0>
```



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.

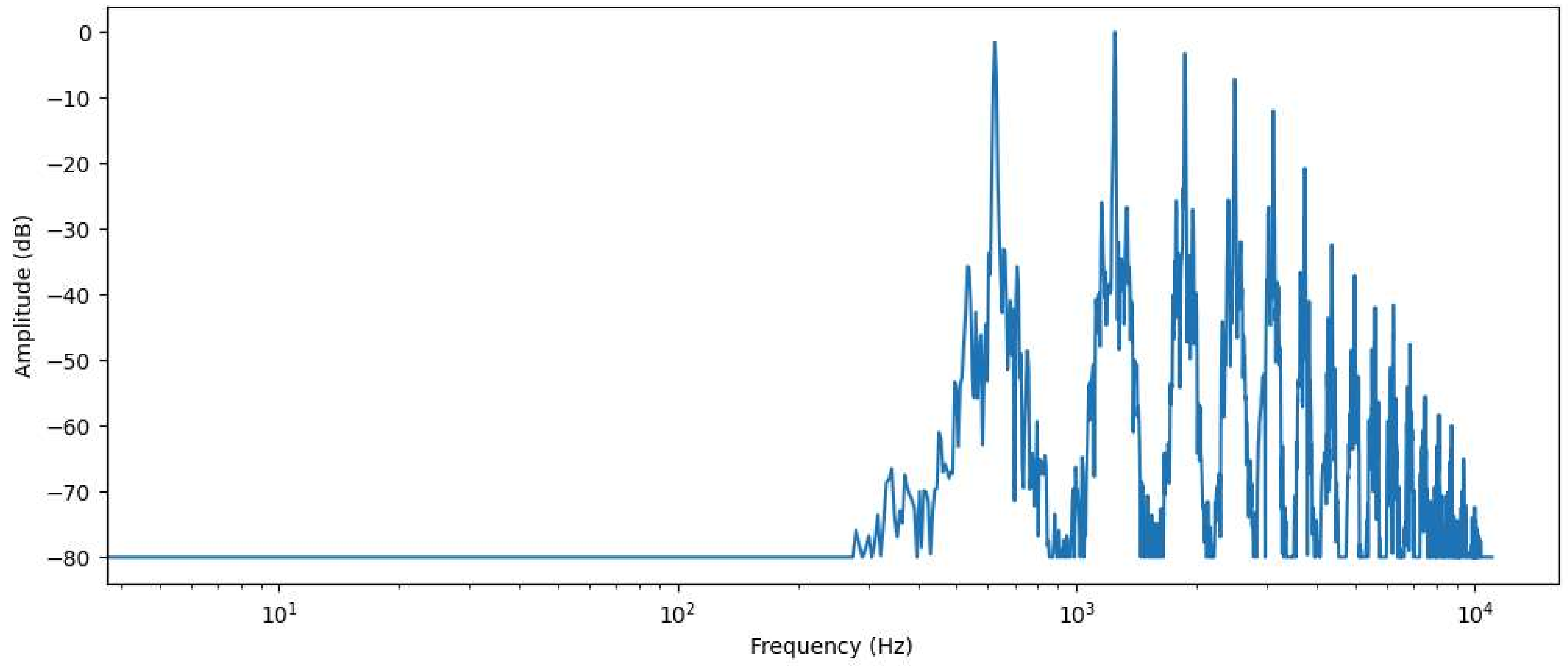
The frequency spectrum

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.

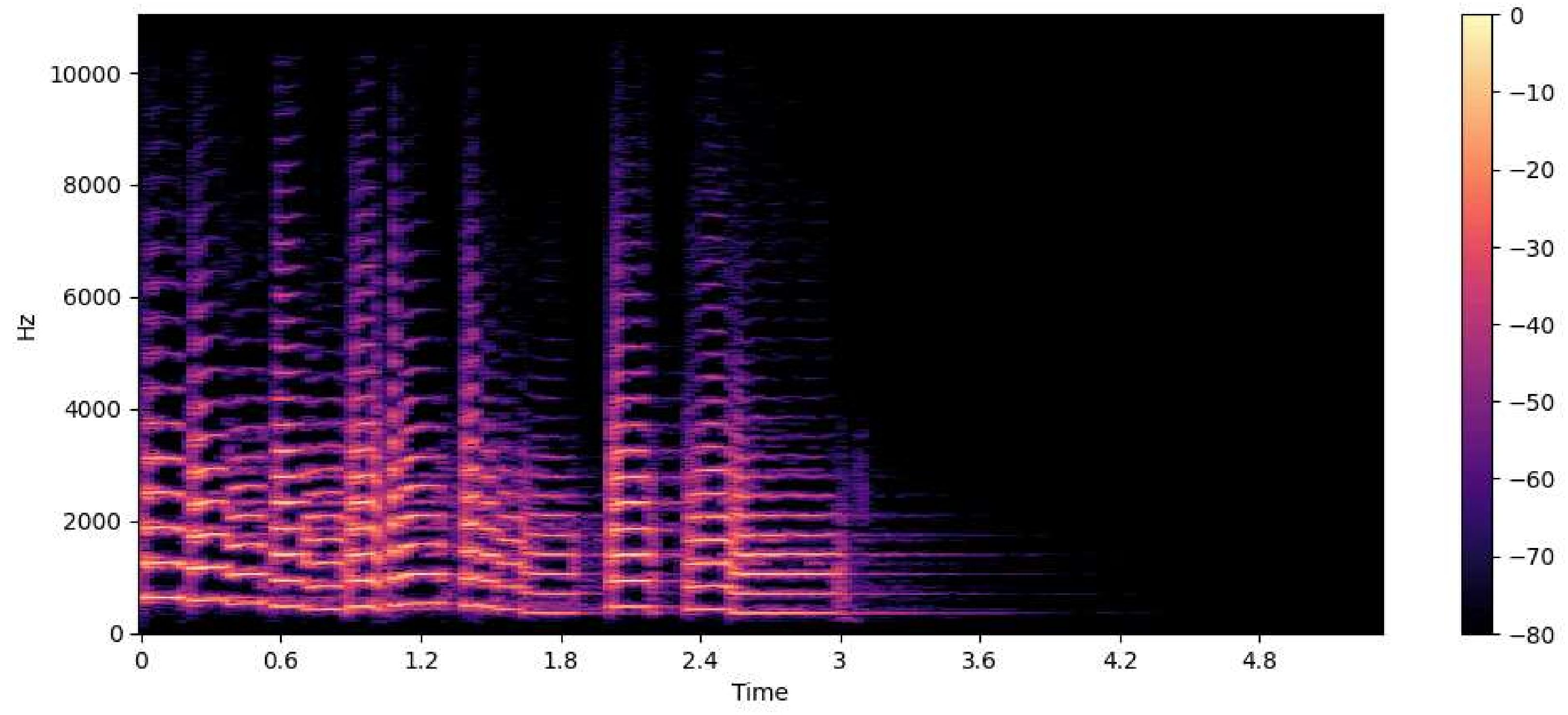
Spectrogram

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.

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.

```
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()
```

```
<matplotlib.colorbar.Colorbar at 0x7f81f0e88a00>
```



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

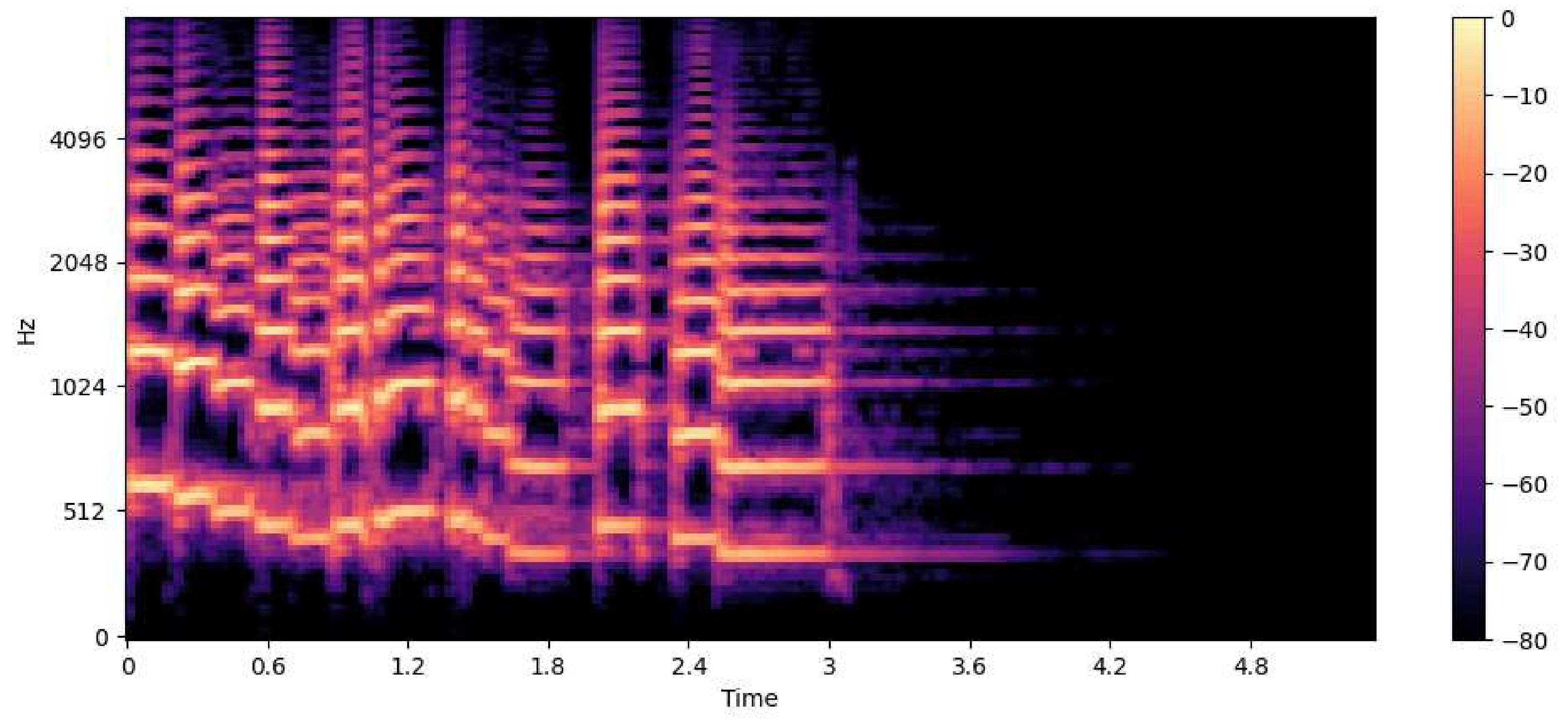
A mel spectrogram is a variation of the spectrogram that is commonly used in speech processing and machine learning tasks. It is similar to a spectrogram in that it shows the frequency content of an audio signal over time, but on a different frequency axis.

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.

```
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=8000)
plt.colorbar()
```

```
<matplotlib.colorbar.Colorbar at 0x7f81ecf91840>
```



n 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.