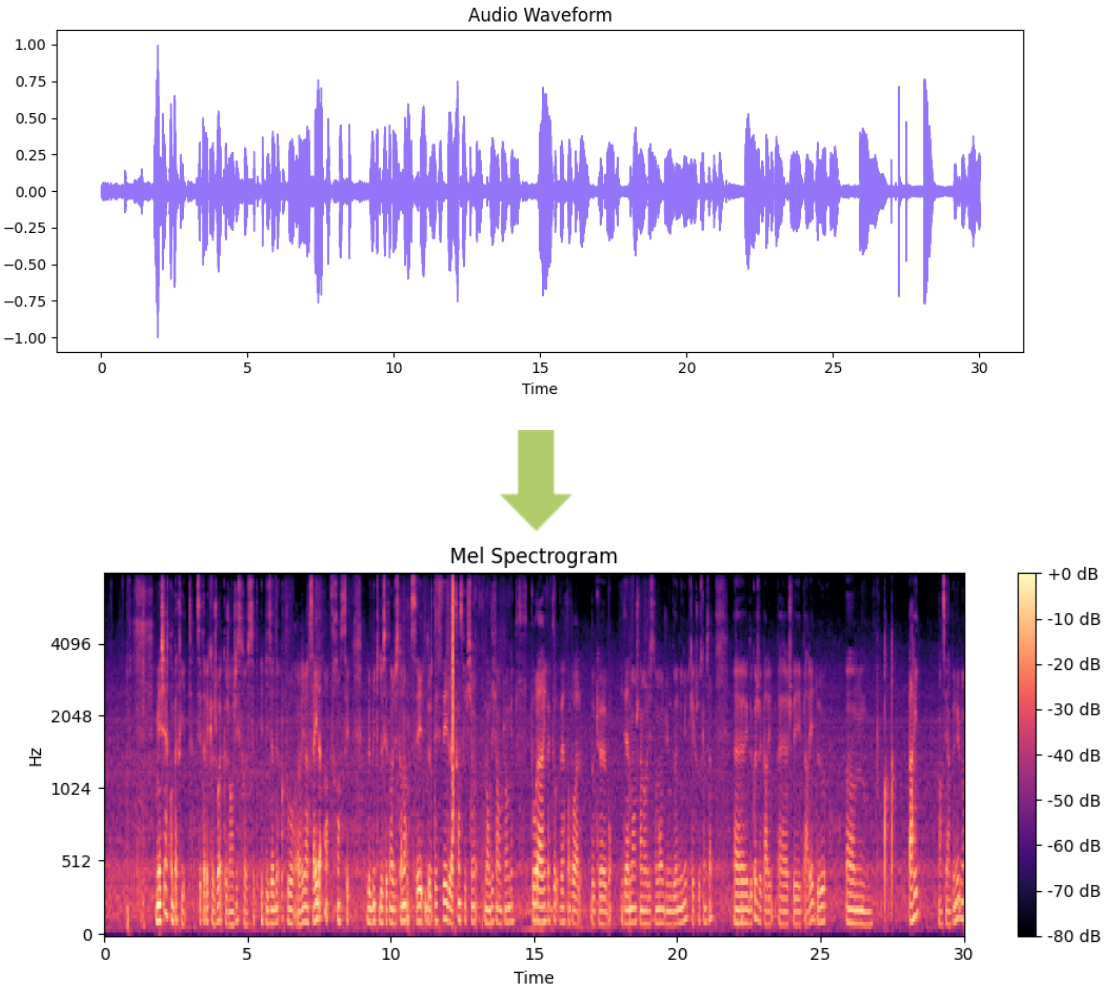
# Riverside.fm - Home assignment

## Product context

At Riverside, we use a speech-to-text model to extract the transcription from a given recording. The speech-to-text model requires pre-processing the recording into Mel-spectrograms.

*What is a Mel-spectrogram ?*

A Mel-spectrogram is a spectral representation of an audio signal on a Mel scale. You don't need to know what it is to complete this assignment, but this is intended to give more insight on the data we use.




The speech-to-text model expects a Mel-spectrogram of shape 80×3000 (80 bins, 100 points per second). If the audio is less than 30 seconds, the spectrogram is padded with zeros to ensure shape consistency.

You can read more about mel-spectrograms here.

These inputs are put in a queue and we want to batch them for optimization purposes.

# BatchQueue

In this assignment you will implement a new class for pytorch called: **BatchQueue**.

This queue differs from a standard queue in the fact that it supports a 'batch' of different queues with a **single tensor**. If the dimensions of a standard queue are [L,F] with L representing the length of the queue and F the number of features, then this queue will have an extra dimension and will have dims [L,B,F] with B representing the batch size.

In our case, F is the number of spectrogram features which we have flatten for ease of use, which means F=240000 (80×3000).
We have provided a file mel_spectrograms.npy that has a shape of (5, 4, 240000) – it is a numpy array of 20 flattened spectrograms, reshaped as 5 batches of 4 elements. This file is used in the provided sanity_check() method to verify your code.

*Please implement the BatchQueue class either in the batch_queue.py file or the Colab notebook BatchQueue.ipynb. The Colab notebook already has pytorch installed.*

**Optimized performance is very important, make sure your implementations is efficient.**

The queue needs to support the following methods:

1. init(dims):
    a. Arguments:
        i. dims : Tensor = 1D tensor of size (3) specifying the dimensions of the queue: [L,B,F]. The dimensions should be treated as follows:
            1. L - queue length
            2. B - batch size
            3. F - number of features
    b. Does not return anything
2. dequeue(batch_indices):
    a. Arguments:
        i. batch_indices : Tensor = 1D tensor of Integers of size (N), 1<=N<=B
    b. Returns a 2D tensor of size (N,F) representing the values in the head of the queues of indices batch_indices. Modifies the queues in indices batch_indices such that each instance is 'moved forward' in line. You may assume the queues are not empty.
3. enqueue(values, batch_indices):
    a. Arguments:
        i. Values : Tensor = 3D tensor of size (T,N,F), 1<=T<=L, 1<=N<=B to be inserted at the tail of the queue of indices [batch_indices]. You may assume the queue has room for the values given.
        i. Batch_indices : Tensor = 1D tensor of Integers of size N, 1<=N<=B
    b. Does not return anything
4. peek(location, batch_indices):
    c. Arguments:
        i. Location : Tensor = list of Strings of length(N), 1<=N<=B that can either be 'head' or 'tail'.
        ii. Batch_indices : Tensor = 1D tensor of Integers of size(N), 1<=N<=B
    d. Returns a 2D tensor of size (N,F) representing the values in the head/tail of the queues at indices batch_indices. You may assume the queues are not empty.

# Visual Example

**dequeue**(batch_indices):

- Batch_queue size=(L=4,B=3,F)
- N=2, batch_indices=torch.tensor([1,2])