Open in Colab

# Data Pre-processing: Audio

In this recitation, we will introduce a few concepts in data preprocessing for audio (speech) that are relevant to the course.

Here is a great reference to read more about speech pre-processing Speech Processing Book

Optional readings:
article on MFCCs

SpecAugment paper

Torch Audio Transforms docs

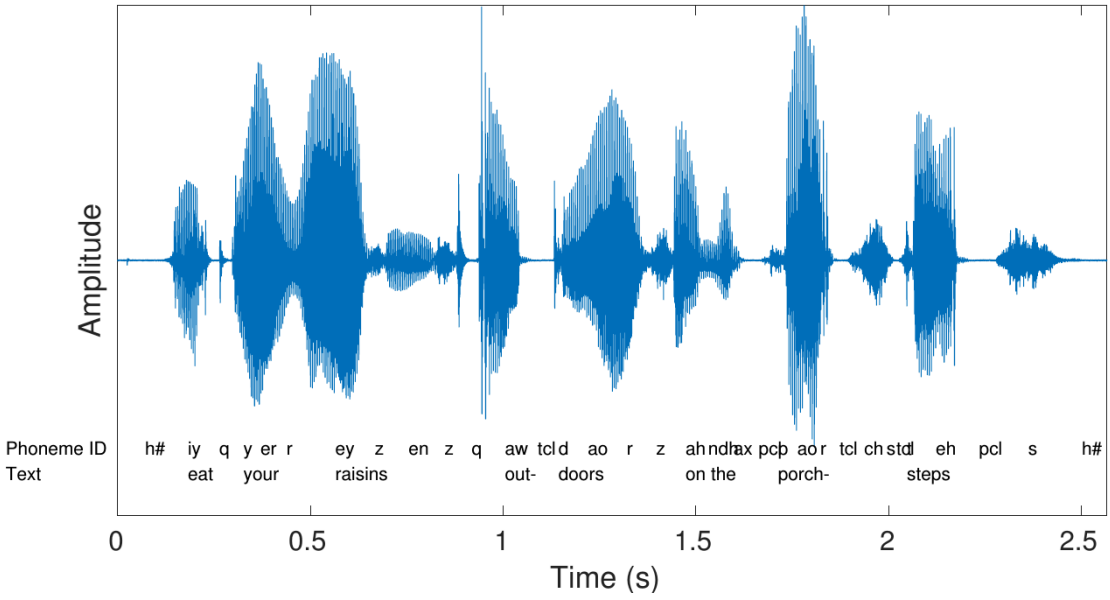
Credit to several semesters of IDL TA's for their contribution to this notebook

## Introduction

### Motivation

Sound signals are pressure variations that are often represented as sound waves that propagate through some medium. In digital systems, these continuous signals are sampled at some (sampling) rate and represented as a $ T \times 1 $ vector where for each time instant, a value of amplitude is recorded. You may think of $ T $ as the total number of samples for the duration of the audio recording $ T = F_s \times duration $ where $ F_s $ is the sampling rate in Hertz.
(Note: we are assuming single channel audio aka mono for this recitation)

Here is an example of a waveform corresponding to some speech, the phonetic (spoken language) and orthographic (written language) representations are also provided.



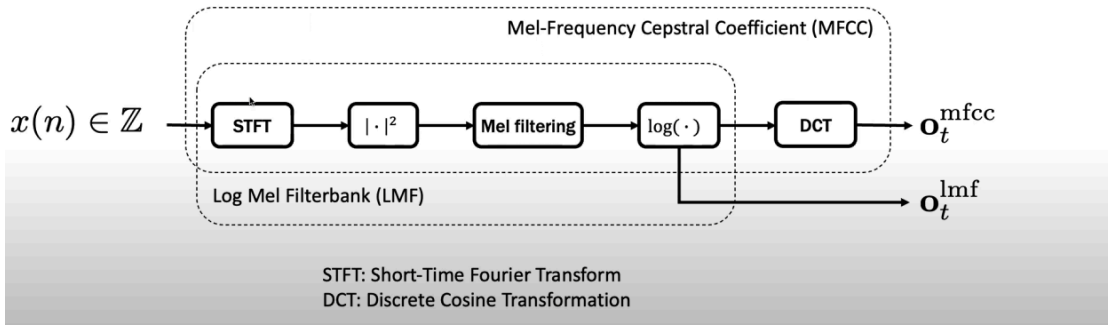

When learning with audio, it is often useful to pre-process the audio input before feeding it into the model. Why?

There are several reasons why that could be the case. Here are a few that are relevant to IDL Homeworks:

- **To learn with dense features**: The $ T \times 1$ ends up being of very **high dimensionality**. For a 2 second utterance sampled at 16kHz, $ T = 16 \times 10^3 \times 2 = 32,000$. It is sometimes desirable to compute features from the raw audio that are more information dense than the audio itself.
- **Augmenting** the dataset to:
  - make the model more robust to challenging cases that might be expected at inference time
  - artificially expand the dataset

There are many other reasons that necessitate data preprocessing.

# Extracting Features: MFCCs




In the homeworks you will come across, the audio datasets will be provided as MFCCs. We will briefly go over the process of extracting MFCCs, but you will not be required to do this during the course.

In [1]:
```
# data loading
import os
import numpy as np
import torch
import torchaudio
import librosa
import librosa.display
import IPython
import IPython.display
import matplotlib.pyplot as plt
import torchaudio.transforms as tat
```

In [2]:
```
!rm *.flac*
# Download some audio files from the LibriSpeech dataset
base_url = "https://dagshub.com/DagsHub/Librispeech-ASR-corpus/raw/2fead7
for i in range(4):
    # only take 1 and 2
    if i not in [1, 2]:
        continue
    file_url = f"{base_url}{i}.flac"
    !wget {file_url}
```

```
zsh:1: no matches found: *.flac*
--2026-02-14 07:15:57--  https://dagshub.com/DagsHub/Librispeech-ASR-corpu
s/raw/2fead768d9690a42d186188ed77a6d4c63c949dd/test-clean/8230/279154/8230
-279154-0001.flac
Resolving dagshub.com (dagshub.com)... 13.107.246.40, 13.107.213.40
Connecting to dagshub.com (dagshub.com)|13.107.246.40|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [application/octet-stream]
Saving to: '8230-279154-0001.flac'

8230-279154-0001.fl     [ <=>                ] 300.04K  --.-KB/s    in 0.1
s

2026-02-14 07:16:00 (2.85 MB/s) - '8230-279154-0001.flac' saved [307246]

--2026-02-14 07:16:01--  https://dagshub.com/DagsHub/Librispeech-ASR-corpu
s/raw/2fead768d9690a42d186188ed77a6d4c63c949dd/test-clean/8230/279154/8230
-279154-0002.flac
Resolving dagshub.com (dagshub.com)... 13.107.213.40, 13.107.246.40
Connecting to dagshub.com (dagshub.com)|13.107.213.40|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [application/octet-stream]
Saving to: '8230-279154-0002.flac'

8230-279154-0002.fl     [ <=>                ] 274.23K  --.-KB/s    in 0.1
s

2026-02-14 07:16:03 (2.52 MB/s) - '8230-279154-0002.flac' saved [280810]
```

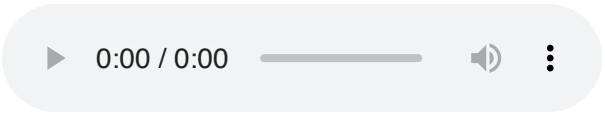
In [14]:
```
# Save the path to the downloaded file
audio_path = "./content/8230-279154-0002.flac"
```

In [15]:
```
type(audio_path), getattr(audio_path, "shape", None)
```

Out[15]:
```
(str, None)
```

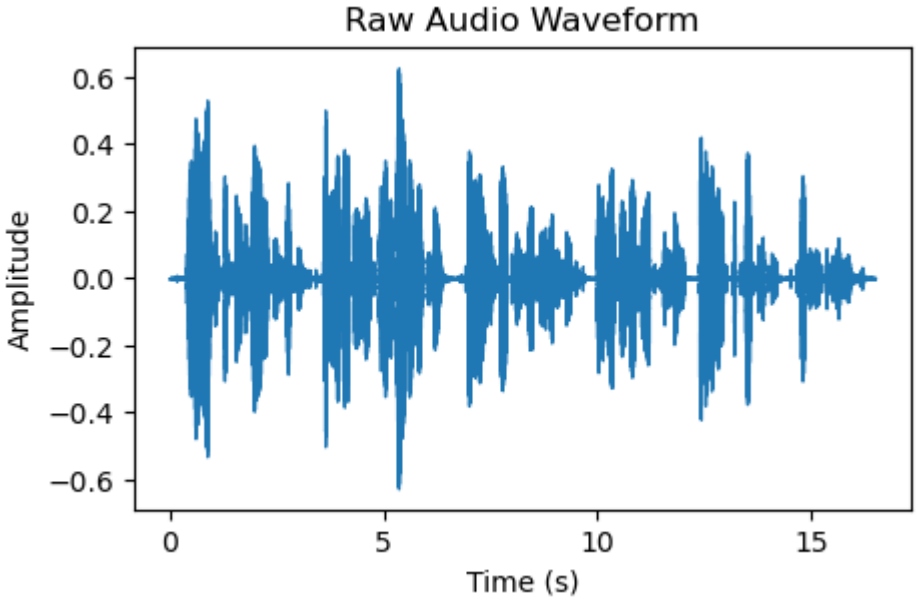
In [16]:
```
# Play the file
IPython.display.Audio(filename=audio_path)
```

Out[16]:




In [17]:
```
# Load the audio file into the and get basic info
waveform, sample_rate = librosa.load(audio_path, sr=None)  # sr=None to k

# Get shape and sample rate
print(f"Waveform shape: {waveform.shape}") #Tells you how many audio samp
print(f"Sample rate: {sample_rate}") #Shows how many data points per seco

# Get length of the audio in seconds
audio_length_seconds = len(waveform) / sample_rate  #Finds the duration o
print(f"Audio length (seconds): {audio_length_seconds}")

# Visualize the raw audio waveform
plt.figure(figsize=(5, 3))
librosa.display.waveshow(waveform, sr=sample_rate)
plt.title('Raw Audio Waveform')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.show()
```

```
Waveform shape: (263600,)
Sample rate: 16000
Audio length (seconds): 16.475
```



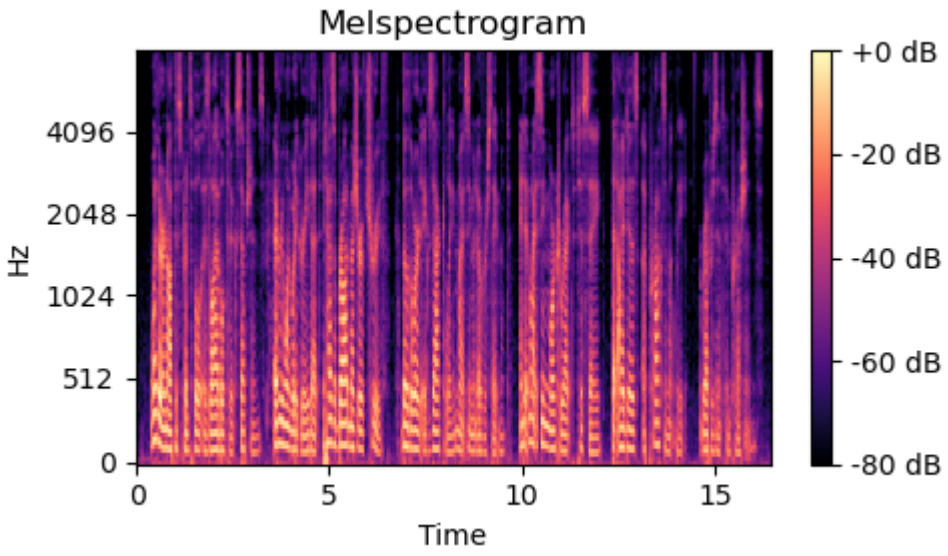

Mel Spectrogram: This is a representation of the audio's energy at various time and frequency bins, using the Mel scale for frequency. It still reflects a detailed "picture" of how energy is distributed across frequencies that matter to human hearing.

Mel Spectrogram:

Used for Visualization and Analysis: Provides a rich, interpretable visualization of the audio's content over time and frequency.

Base for Other Features: Many downstream features (like MFCCs) are computed from the mel spectrogram.

In [18]:
```
# Convert raw audio --> melspectrogram
S = librosa.feature.melspectrogram(y=waveform, sr=sample_rate) #
S_dB = librosa.power_to_db(S, ref=np.max) #The spectrogram values, origin
# This makes the values easier to interpret visually and aligns better wi

# Visualize the melspectrogram
def plot_melspectrogram(S_dB, sample_rate, title="Melspectrogram"):
    plt.figure(figsize=(5, 3))
    librosa.display.specshow(S_dB, sr=sample_rate, x_axis='time', y_axis=
    plt.colorbar(format='%+2.0f dB')
    plt.title(title)
    plt.tight_layout()
    plt.show()

plot_melspectrogram(S_dB, sample_rate)
```




MFCCs (Mel-Frequency Cepstral Coefficients): These are further-processed features derived from the mel spectrogram by applying a Discrete Cosine Transform (DCT) to compress and decorrelate the information. MFCCs summarize the spectral envelope in a compact way, leading to features that are highly effective for speech and audio tasks.

MFCCs:

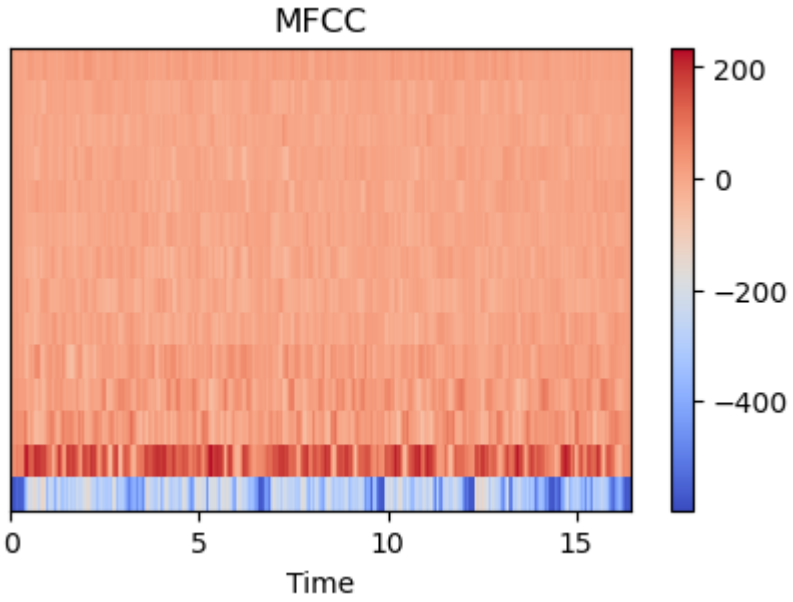
Compact Feature Representation: MFCCs reduce the dimensionality and redundancy of the mel spectrogram, yielding a set of coefficients that summarize each timestep.

Better for ML Models: These are more “dense” and robust features, less sensitive to non-relevant variance, making them ideal for feeding into machine learning models.

In [19]:
```
# Convert raw audio --> mel-frequency cepstral coefficients (MFCC)
mfccs = librosa.feature.mfcc(y=waveform, sr=sample_rate, n_mfcc=14) # Con

# print shape
print(f"MFCC shape: {mfccs.shape}") # Displays the dimensions of the MFCC
```

```
# Visualize the MFCC
plt.figure(figsize=(5, 3))
librosa.display.specshow(mfccs, sr=sample_rate, x_axis='time')
plt.colorbar()
plt.title('MFCC')
plt.show()
```

```
MFCC shape: (14, 515)
```

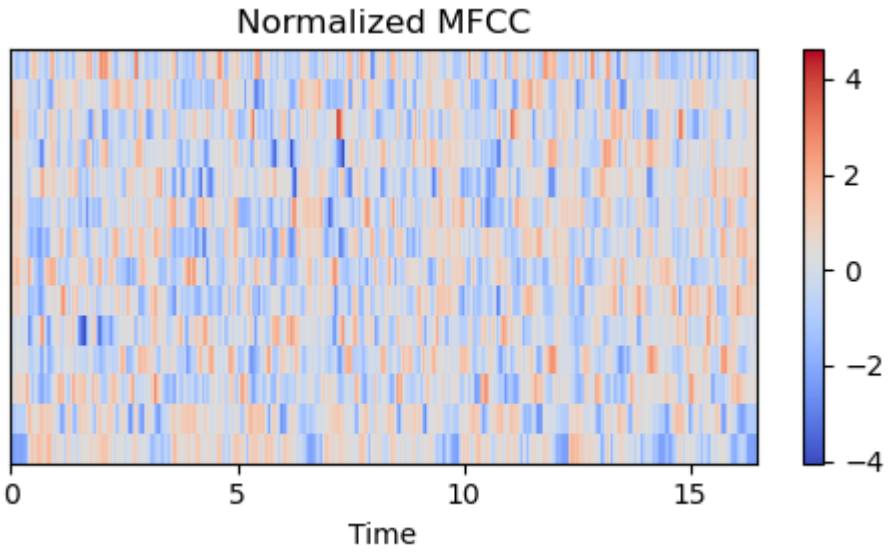



## Cepstral Normalization

A process of scaling the MFCCs so that they have a mean of zero and a standard deviation of one. This standardization technique makes sure that the MFCCs are not dominated by some particular part of the spectrum and that all features contribute equally to the analysis. This has several potential benefits:

- reduction of noise impact
- better robustness across environments
- enhancement of feature discriminability

In [20]:
```
# Apply normalization
mfccs_normalized = (mfccs - mfccs.mean(axis=1, keepdims=True)) / (mfccs.s

# Visualize MFCCs after cepstral normalization
plt.figure(figsize=(5, 3))
librosa.display.specshow(mfccs_normalized, sr=sample_rate, x_axis='time')
plt.colorbar()
plt.title('Normalized MFCC')
plt.tight_layout()
plt.show()
```




# Augmentations for MFCCs

These are motivated by the SpecAugment paper. The objectives are to:

- reducing overfitting
- improve robustness to deformations in the time direction, partial loss of frequency information and partial loss of small segments of speech

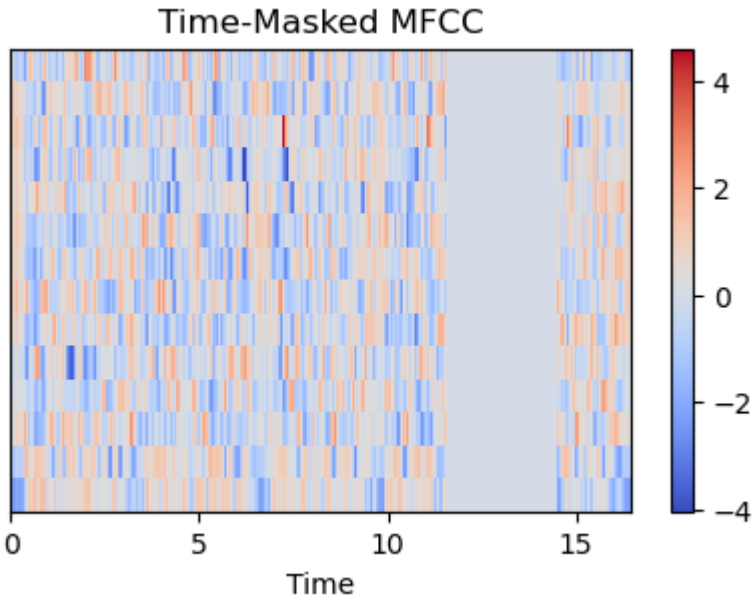
## Time masking

parameters:

- **time_mask_param**: maximum possible length of the mask. Indices uniformly sampled from [0, time_mask_param).
- **iid_masks**: whether to apply different masks to each example/channel in the batch.
- **p**:maximum proportion of time steps that can be masked. Must be within range [0.0, 1.0]

In [21]:
```
time_mask = tat.TimeMasking(time_mask_param=100, p=0.2)
#time_mask_param=100: The maximum possible length (in frames) for a mask
#p=0.2: The probability or proportion of the time axis that can be masked
time_masked_mfcc = time_mask(torch.tensor(mfccs_normalized))
# plot
plt.figure(figsize=(5, 3))
librosa.display.specshow(time_masked_mfcc.numpy(), sr=sample_rate, x_axis
plt.colorbar()
plt.title('Time-Masked MFCC')
```

Out[21]:
```
Text(0.5, 1.0, 'Time-Masked MFCC')
```



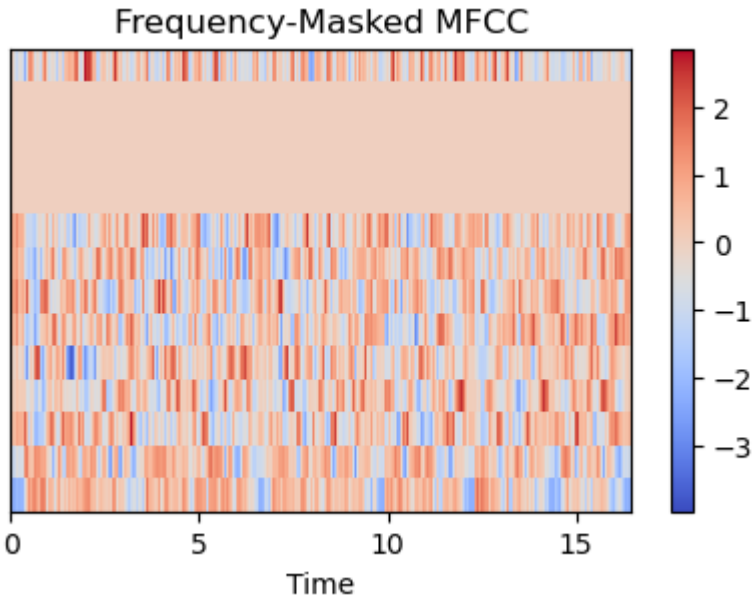

## Frequency masking

parameters:

- **freq_mask_param**: maximum possible length of the mask. Indices uniformly sampled from [0, freq_mask_param).
- **iid_masks**: whether to apply different masks to each example/channel in the batch

In [22]:
```
freq_mask = tat.FrequencyMasking(freq_mask_param=5) #sets the maximum pos
freq_masked_mfcc = freq_mask(torch.tensor(mfccs_normalized))
# plot
plt.figure(figsize=(5, 3))
librosa.display.specshow(freq_masked_mfcc.numpy(), sr=sample_rate, x_axis
plt.colorbar()
plt.title('Frequency-Masked MFCC')
```

Out[22]:
```
Text(0.5, 1.0, 'Frequency-Masked MFCC')
```



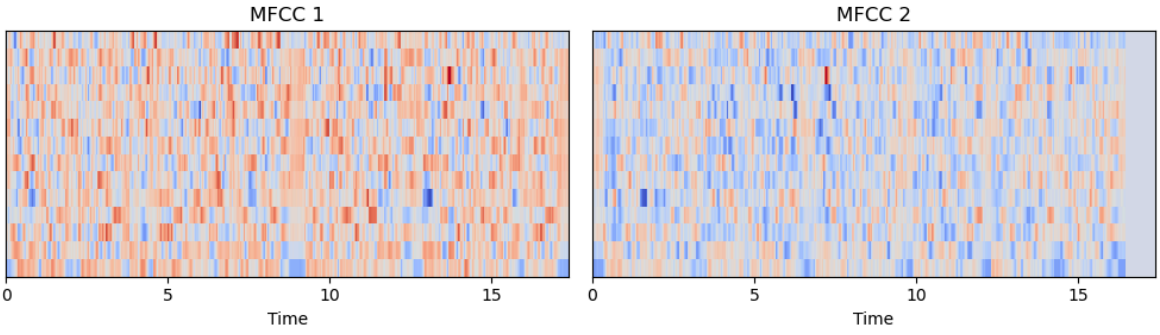
Augmentations on a mini-batch

In [24]:
```
import glob
audio_files = glob.glob('./content/8230-279154-000*.flac') #finding all t
print(audio_files)
```

```
['./content/8230-279154-0001.flac', './content/8230-279154-0002.flac']
```

The next code block processes a batch of audio files to extract and visualize their MFCC (Mel-Frequency Cepstral Coefficient) features in a standardized, comparative way.

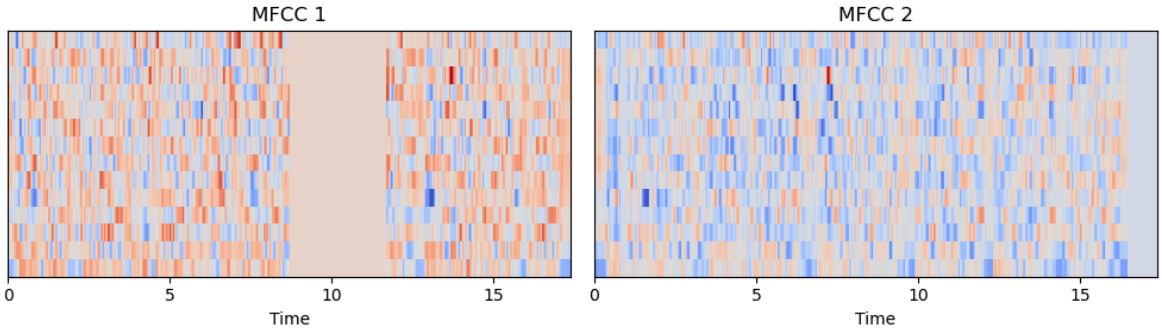
In [25]:
```
# load all audio files
waveforms = []
for audio_file in audio_files:
    waveform, sample_rate = librosa.load(audio_file, sr=None)
    waveforms.append(waveform)

# Compute MFCCs
mfccs_batch = []
for waveform in waveforms:
    mfcc = librosa.feature.mfcc(y=waveform, sr=sample_rate, n_mfcc=14)
    # normalize them
    mfcc = (mfcc - mfcc.mean(axis=1, keepdims=True)) / (mfcc.std(axis=1,
    mfccs_batch.append(mfcc)

# Pad MFCCs to the same length
max_length = max(mfcc.shape[1] for mfcc in mfccs_batch)
padded_mfccs_batch = [np.pad(mfcc, ((0, 0), (0, max_length - mfcc.shape[1

# Plot as a 2 by 2 grid
fig, axes = plt.subplots(1, 2, figsize=(10, 3))
axes = axes.flatten()

for i, mfcc in enumerate(padded_mfccs_batch):
    if i < 4:  # Ensure there are only 4 plots
        ax = axes[i]
        librosa.display.specshow(mfcc, sr=sample_rate, x_axis='time', ax=
        ax.set_title(f'MFCC {i+1}')

plt.tight_layout()
plt.show()
```




In [26]:
```
# time masking in a batch with IID mask = true
time_mask = tat.TimeMasking(time_mask_param=100, iid_masks=True, p=0.8)
time_masked_mfccs_batch = time_mask(torch.tensor(padded_mfccs_batch))
# plot
# Plot as a 2 by 2 grid
```

```
fig, axes = plt.subplots(1, 2, figsize=(10, 3))
axes = axes.flatten()

for i, mfcc in enumerate(time_masked_mfccs_batch):
    if i < 4:  # Ensure there are only 4 plots
        ax = axes[i]
        librosa.display.specshow(mfcc.numpy(), sr=sample_rate, x_axis='ti
        ax.set_title(f'MFCC {i+1}')

plt.tight_layout()
plt.show()
```

```
/var/folders/5t/4lvt4xjx3ms90skxrzj3r0f40000gn/T/ipykernel_1999/414558016
6.py:3: UserWarning: Creating a tensor from a list of numpy.ndarrays is ex
tremely slow. Please consider converting the list to a single numpy.ndarra
y with numpy.array() before converting to a tensor. (Triggered internally
at /private/var/folders/k1/30mswbxs7r1g6zwn8y4fyt500000gp/T/abs_951ooclnw
1/croot/libtorch_1746637517770/work/torch/csrc/utils/tensor_new.cpp:281.)
  time_masked_mfccs_batch = time_mask(torch.tensor(padded_mfccs_batch))
```




In [27]:
```
# frequency masking with IID = true
freq_mask = tat.FrequencyMasking(freq_mask_param=4, iid_masks=True)
freq_masked_mfccs_batch = freq_mask(torch.tensor(padded_mfccs_batch))
# plot
# Plot as a 2 by 2 grid
fig, axes = plt.subplots(1, 2, figsize=(10, 3))
axes = axes.flatten()

for i, mfcc in enumerate(freq_masked_mfccs_batch):
    if i < 4:  # Ensure there are only 4 plots
        ax = axes[i]
        librosa.display.specshow(mfcc.numpy(), sr=sample_rate, x_axis='ti
        ax.set_title(f'MFCC {i+1}')

plt.tight_layout()
plt.show()
```