AUDIO FEATURES

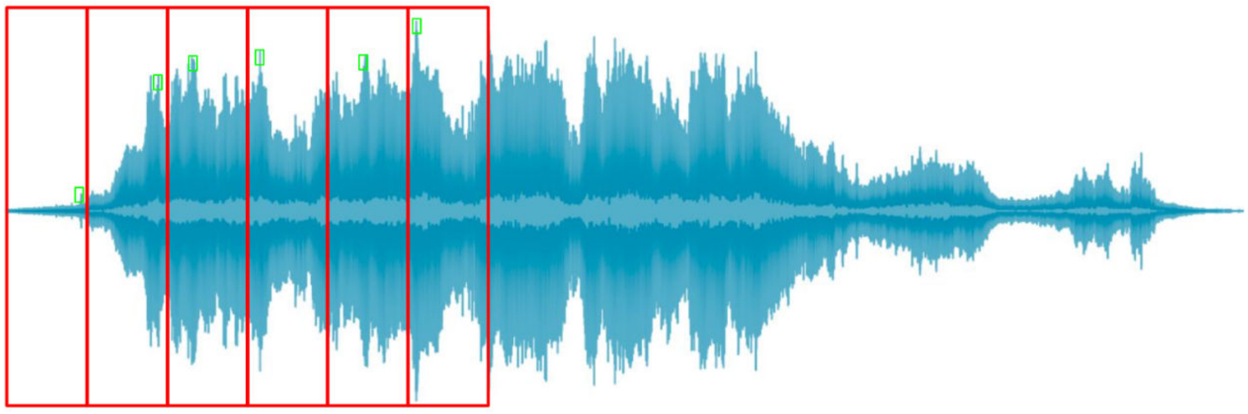
ARIHARASUDHAN

⁕ THE BABBLER

A personification exists for individuals who tend to speak prolifically, likening them to the Jungle Babbler Bird. This bird is known for laying distinctive blue eggs. When confronted by a snake or other predators, the bird emits warning calls, drawing the threat towards its nest. By remaining silent and blending into the foliage, the bird may have avoided attracting predators. The folk saying, "Control your mouth or you'll be stuck like a babbler," underscores the importance of discretion. These anecdotes are centered around AUDIO.

⁕ AUDIO FEATURES

In Speech Processing, the main step to be considered the most is, “EXTRACTING FEATURES CORRECTLY”. If this step is done correctly, yeah hoo! Different types of features are used in ASR systems, each with its own characteristics and advantages. Some of them are summarized in this small book by Ari.



✦ Acoustic Features

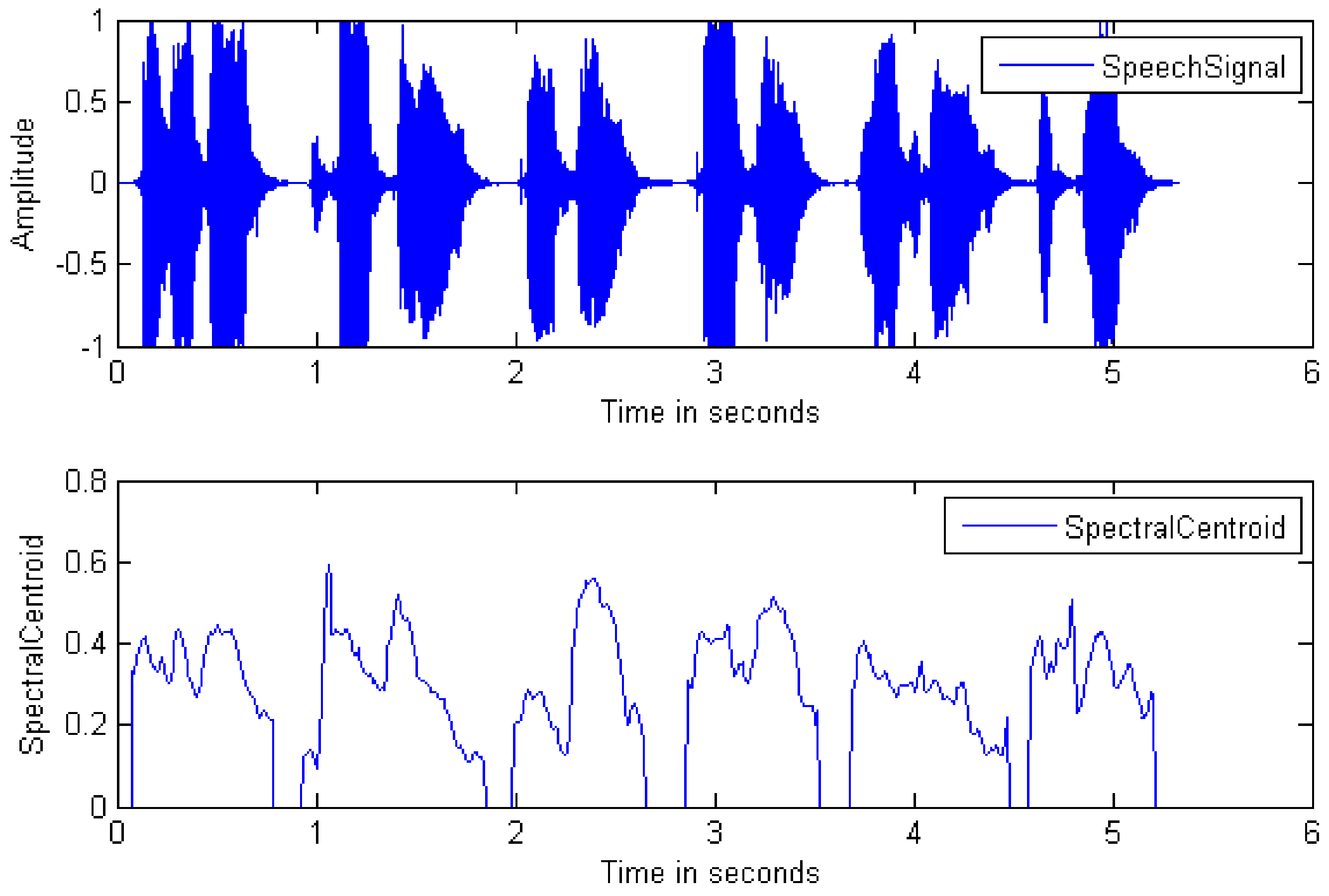
The word, “acoustic” has a Greek origin of “akouein” which means, “to hear”. Acoustic features refer to various characteristics of sound signals that can be measured and analyzed. Some common acoustic features are Pitch, Intensity, Duration, Timbre, Spectral Features, Mel-Frequency Spectral Coefficients, HNR Ratio, Rhythm and so on.

1. Pitch: The perceived frequency of a sound, which determines whether it is high or low. (Hz)

2. Intensity: The strength or loudness of a sound wave, measured in decibels (dB).

3.Duration: The length of time a sound persists, typically measured in seconds.

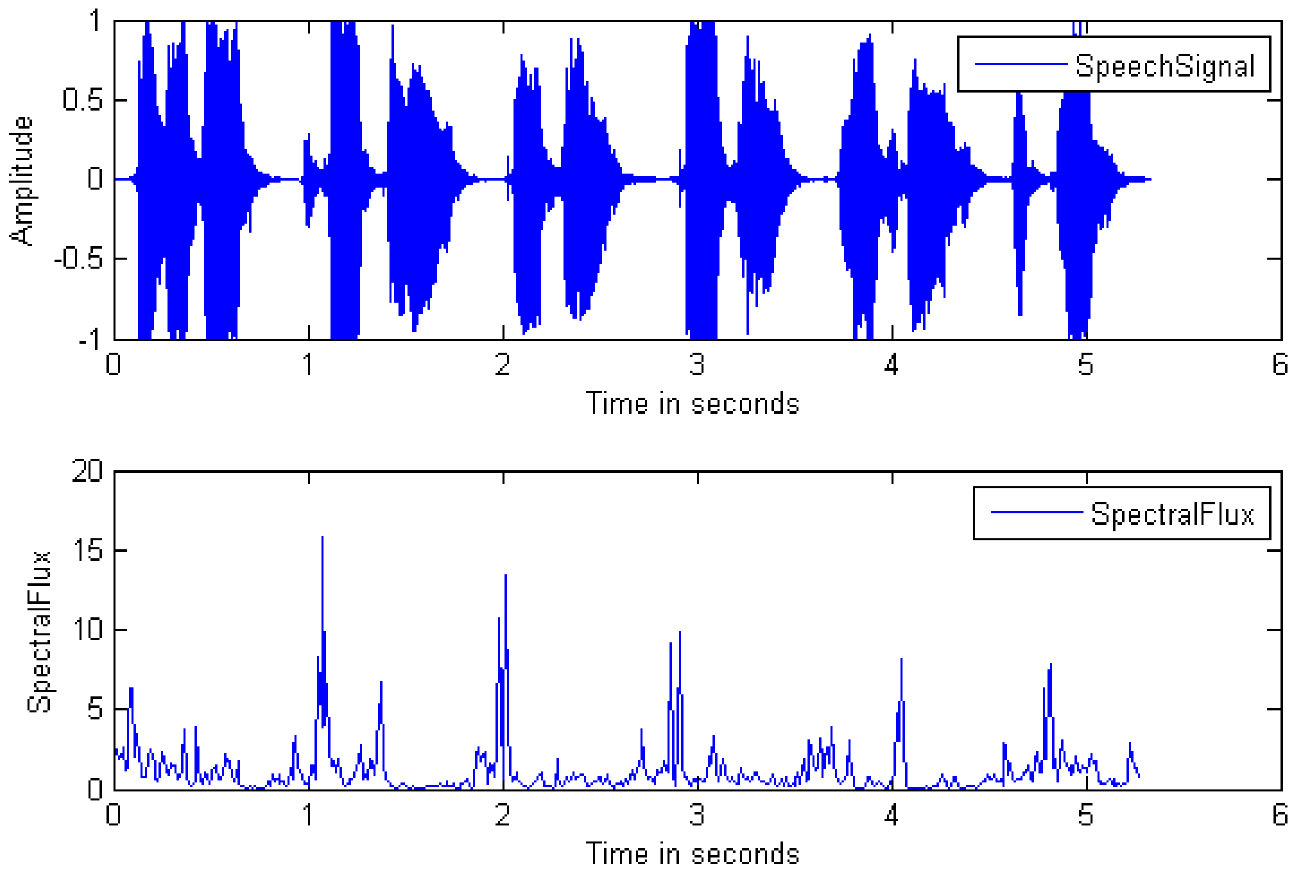
4.Timbre: The quality or color of a sound that distinguishes it from other sounds with the same pitch and loudness.



5.Spectral Features

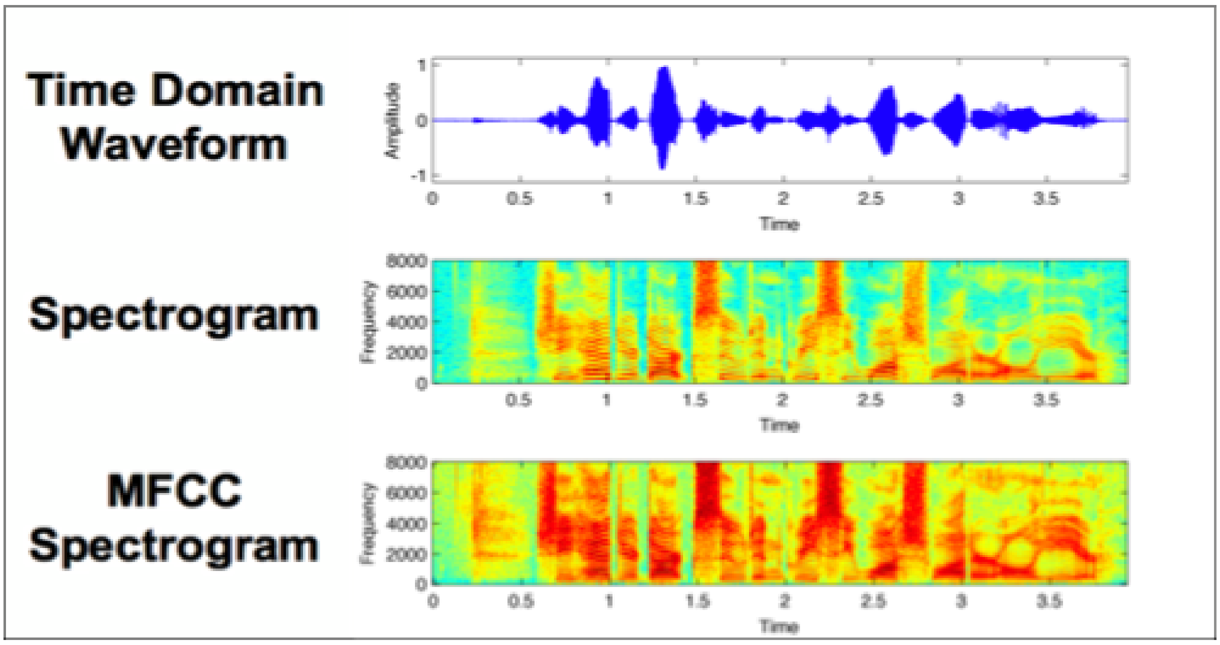
>Spectral Centroid: The center of mass of the spectrum, indicating the "brightness" of a sound.

>Spectral Flux: A measure of how quickly the spectrum of a signal is changing over time.

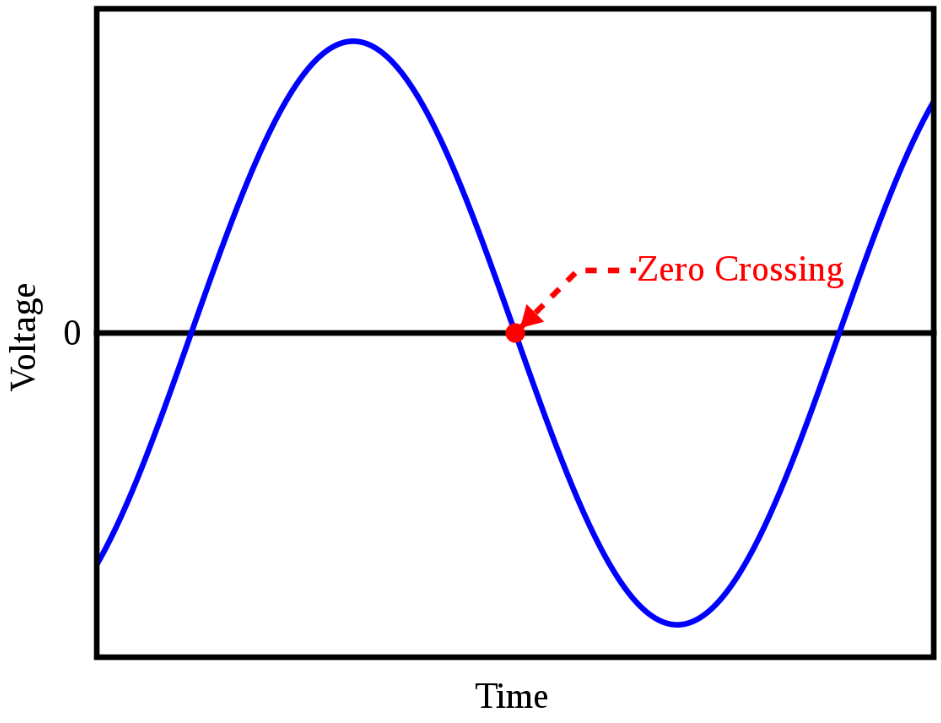


>Spectral Roll-Off: The frequency below which a certain percentage of the total spectral energy is contained.

6.Mel-frequency Cepstral Coefficients: MFCCs are a set of features used to represent the characteristics of a sound signal. These are a way of capturing essential information about the frequency content of a sound signal in a format that is more suitable for human hearing and efficient for various signal processing tasks. The Mel scale is a way to measure the perceived pitch or frequency of sounds, mimicking how humans hear different pitches. It's adjusted to be more in line with human hearing, giving more emphasis to lower frequencies. The spectrum of a sound is a set of different frequency bands. MFCCs take this spectrum and divide it into specific bins or segments, each corresponding to a different range of frequencies. Each of these bins is then filtered using triangular filters. These filters are designed to mimic how our ears respond to different frequencies. After filtering, the amplitudes of the filtered signals are transformed using a logarithmic function. This is done because human hearing is more sensitive to changes in loudness at lower volumes, and a logarithmic scale better represents this sensitivity. The final step involves applying a mathematical transformation called the Discrete Cosine Transform (DCT). This step helps in decorrelating the values obtained from the logarithmic compression, resulting in a set of coefficients that can effectively represent the unique characteristics of the sound.

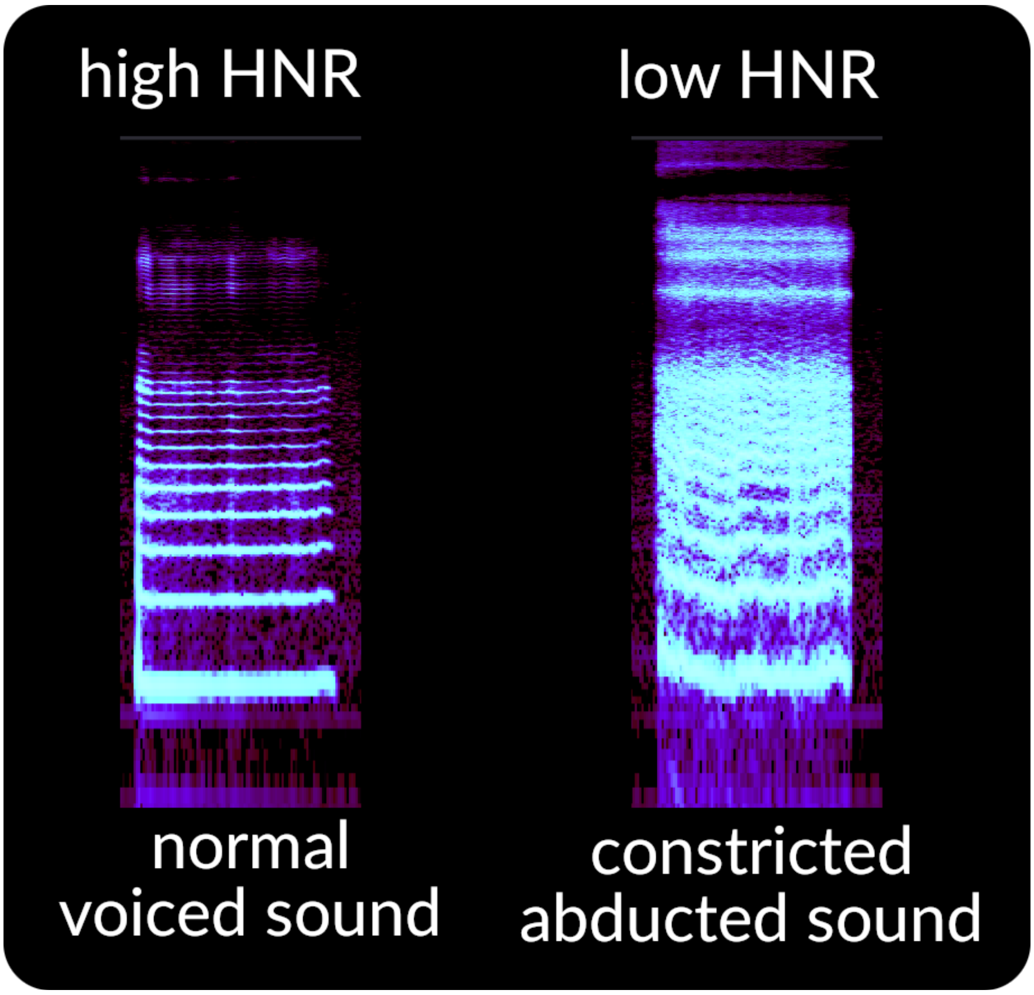


6.Zero-Crossing Rate: The rate at which the signal changes its sign, often used in speech and music analysis.



7. Rhythm/Tempo: The pattern of beats or time intervals in a sequence of sounds, commonly associated with music analysis.

8. Harmonic-to-Noise Ratio: A measure of the proportion of harmonic components to noise components in a sound signal.



✦ Pre-Emphasis Filtering

Pre-emphasis is a filtering technique applied to audio signals before they are transmitted or processed. Its primary purpose is to amplify higher-frequency components relative to lower-frequency components. This is done to counteract the natural attenuation of high-frequency components that occurs during the recording or transmission of audio signals. The pre-emphasis process typically involves boosting the higher frequencies by applying a high-pass filter to the signal. Mathematically, the pre-emphasis operation can be represented by the following equation:

Y(t) = X(t) – alpha.X(t-1)

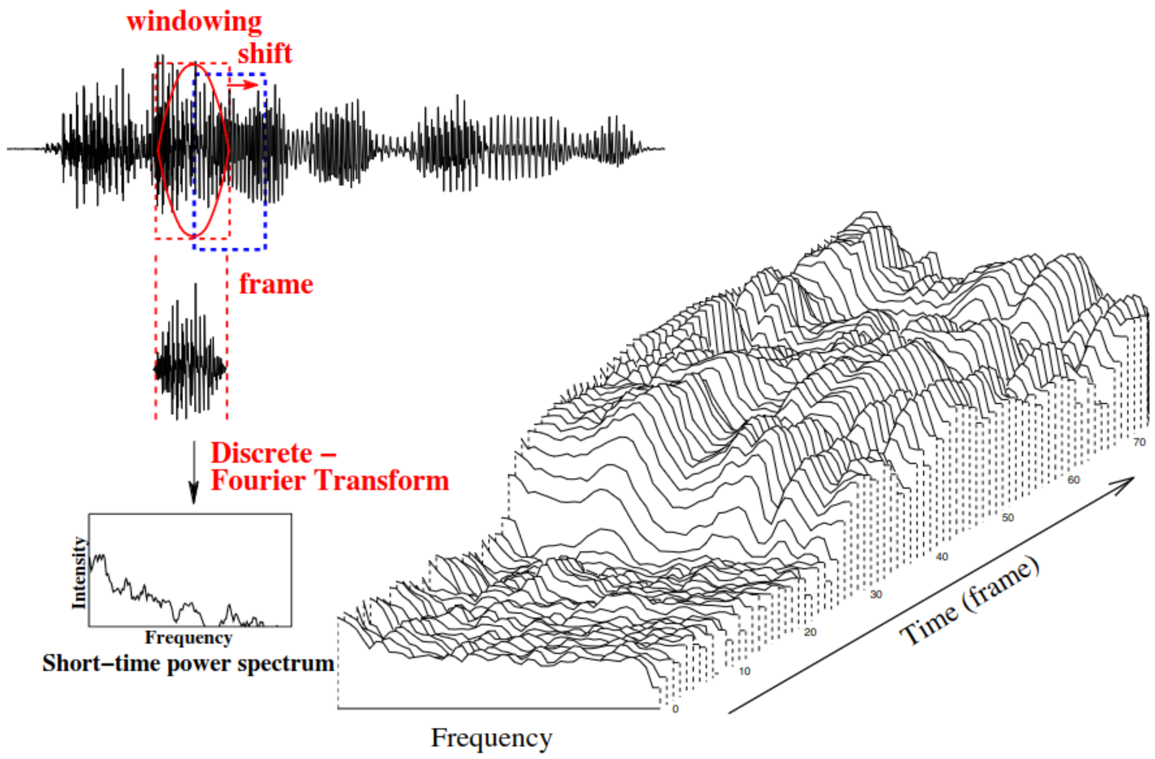
Y(t) is the output signal after pre-emphasis. X(t) is the input signal. alpha is the pre-emphasis coefficient (value typically between 0.9 and 1.0). Without pre-emphasis, the high-frequency components may be attenuated during recording or transmission. By applying pre-emphasis, we can boost the high-frequency components, improving the signal-to-noise ratio.

✦ Spectral Tilt

Spectral tilt refers to the balance between high and low frequencies in a sound signal's spectrum. It is a measure of the distribution of energy across different frequency bands. A signal is said to have a positive spectral tilt if higher frequencies are more prominent than lower frequencies, and a negative spectral tilt if the lower frequencies dominate. One way to compute the spectral tilt is by comparing the energy in high-frequency bands to that in low-frequency bands.

For example, we might calculate the ratio of energy in the high-frequency range to that in the low-frequency range.

✦ Windowing



Audio Signals are often not constant over time. They change their characteristics. Windowing is a technique used to analyze a signal in short, overlapping segments, assuming that the signal is stationary within each segment. The signal is divided into short, overlapping sections called windows. Each window typically has a fixed duration and is chosen to capture the relevant information of the signal within that time frame. Each window is multiplied element-wise with a window function.This multiplication helps to reduce artifacts at the edges of the window and minimize the impact of abrupt changes.

By analyzing the signal in short, overlapping segments, we can better capture the local characteristics and changes in the signal over time.

✦ FFT

Spectral analysis involves examining the frequency content of a signal. It helps to understand which frequencies are present in the signal and their respective amplitudes. One of the common methods for spectral analysis is to apply the Fast Fourier Transform (FFT) to each windowed segment.

FFT converts the signal from the time domain to the frequency domain, revealing the frequency components present in that particular time window. The result of the FFT is often represented as a power spectrum, which shows the distribution of signal power across different frequencies. It provides information about the strength or amplitude of each frequency component. By analyzing multiple windowed segments and their spectra, we can track how the frequency content of the signal changes over time.

✦ DFT

The Discrete Fourier Transform (DFT) is a mathematical operation that transforms a sequence of complex numbers representing a discrete-time signal into another sequence of complex numbers representing the signal's frequency components. The DFT operates on a sequence of discrete data points, typically representing a time-domain signal. Let's say we have a sequence of N complex numbers, denoted as x[n], where n is an index representing time. The DFT transforms this sequence into another sequence of complex numbers, often referred to as X[k], where k is an index representing the frequency components. Each X[k] value represents the amplitude and phase of a sinusoidal component at a specific frequency in the original signal.

Discrete Fourier Transform (DFT):

$$X[k] = \sum_{n=0}^{N-1} x[n] \exp\left(-j\frac{2\pi}{N}kn\right)$$

NB: $\exp(j\theta) = e^{j\theta} = \cos(\theta) + j\sin(\theta)$

The output of the DFT consists of complex numbers. The real part represents the amplitude, and the imaginary part represents the phase of each frequency component.

✦ IDFT

There is also an inverse operation called the Inverse Discrete Fourier Transform (IDFT) that can reconstruct the original signal from its frequency components. The DFT is often computed efficiently using algorithms such as the Fast Fourier Transform (FFT), which significantly reduces the number of computations required compared to the direct application of the DFT formula. The DFT and its variations are essential tools in various applications, including audio signal processing, image processing, communications, and more.

✦ Wide and Narrow Bands

Wide-band and narrow-band spectrograms refer to two different types of visual representations of the frequency content of a signal over time. A wide-band spectrogram displays a broad range of frequencies. It provides a detailed view of the entire frequency spectrum of a signal, typically covering a wide range from low to high frequencies. A narrow-band spectrogram focuses on a specific or narrow range of frequencies. It zooms in on a smaller portion of the frequency spectrum to provide a more detailed view of the signal within that frequency range. Both wide-band and narrow-band spectrograms are created by applying the Short-Time Fourier Transform (STFT) to a signal, which involves dividing the signal into short overlapping segments and computing the Fourier Transform for each segment.

The STFT results in a time-frequency representation where time is along one axis, frequency along the other axis, and the color/intensity represents the magnitude or power of the signal at different time-frequency points.

✦ An Implementation

This example uses the commands dataset and spectrogram as features. The extracted spectrogram images are fed to a CNN and a cross entropy loss is applied. Evidently, It is a classification problem. As we know, Spectrograms provide a visual representation of the frequency content over time, which is useful for capturing the spectral characteristics of speech.

You can get the following notebook from this link

```
import os
import torch
import torchaudio
import matplotlib.pyplot as plt
from torch.utils.data import Dataset, DataLoader
from pathlib import Path
```

LOAD AUDIO

load_audio_files() takes a file path and a label as inputs. It reads WAV audio files from the specified path, extracting information. Using torchaudio for audio loading, the function organizes these details into a list, making it useful for tasks such as speech recognition or classification.

```
def load_audio_files(path: str, label:str):
    dataset = []
    walker = sorted(str(p) for p in Path(path).glob(f'*.wav'))
    for i, file_path in enumerate(walker):
        path, filename = os.path.split(file_path)
        speaker, _ = os.path.splitext(filename)
        speaker_id, utterance_number = speaker.split("_nohash_")
        utterance_number = int(utterance_number)
        waveform, sample_rate = torchaudio.load(file_path)
        dataset.append([waveform, sample_rate, label, speaker_id, utterance_number])
    return dataset
```

```
trainset_speechcommands_tree = load_audio_files('/kaggle/working/SpeechCommands/speech_commands_v0.02/tree', 'tre
trainset_speechcommands_cat = load_audio_files('/kaggle/working/SpeechCommands/speech_commands_v0.02/cat', 'cat')
```

DATALOADERS

```
trainloader_tree = torch.utils.data.DataLoader(trainset_speechcommands_tree, batch_size=1,
                                               shuffle=True, num_workers=0)
trainloader_cat = torch.utils.data.DataLoader(trainset_speechcommands_cat, batch_size=1,
                                               shuffle=True, num_workers=0)
```

WAVEFORMS

A waveform refers to the graphical representation of sound signal. It depicts how the air pressure varies over time.

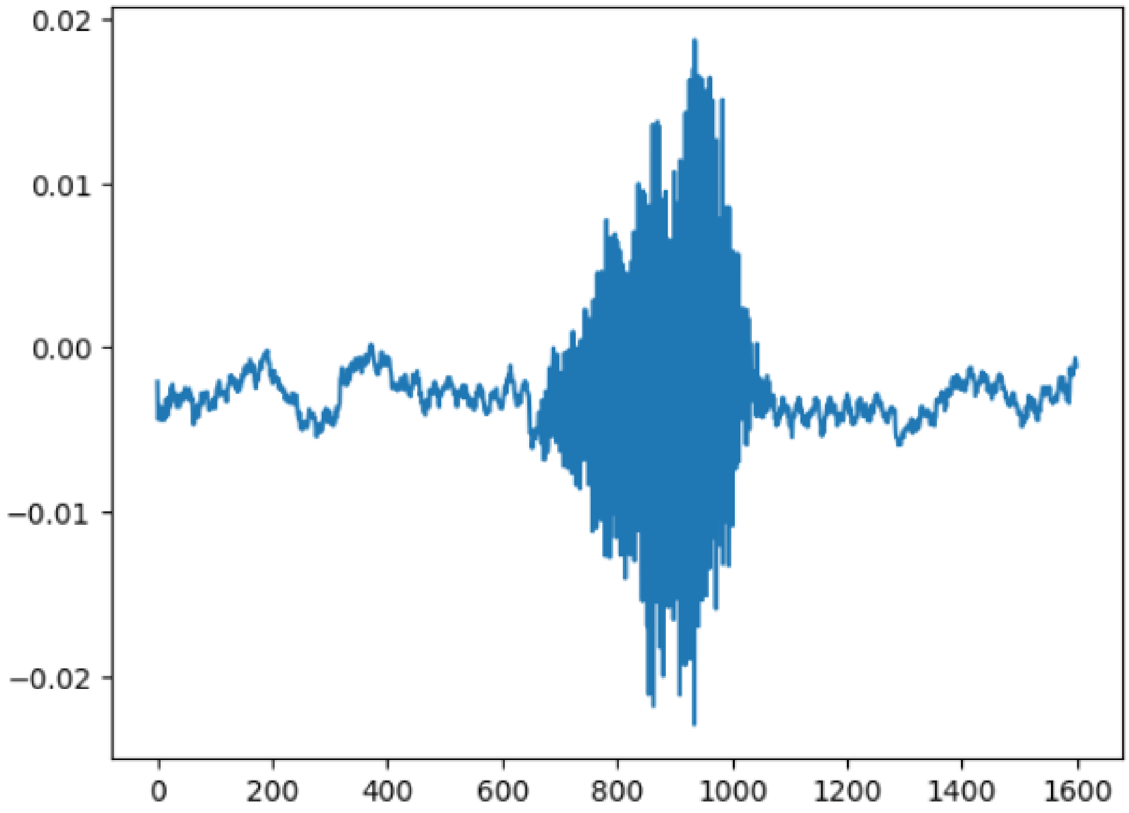
```
def show_waveform(waveform, sample_rate, label):
    print("Waveform: {}\nSample rate: {}\nLabels: {} \n".format(waveform, sample_rate, label))
    new_sample_rate = sample_rate/10
    channel = 0
    waveform_transformed = torchaudio.transforms.Resample(sample_rate, new_sample_rate)(waveform[channel,:].view(
    print("Shape of transformed waveform: {}\nSample rate: {}".format(waveform_transformed.size(), new_sample_rat
    plt.figure()
    plt.plot(waveform_transformed[0,:].numpy())
```

```
cat_waveform = trainset_speechcommands_cat[0][0]
cat_sample_rate = trainset_speechcommands_cat[0][1]
tree_waveform = trainset_speechcommands_tree[0][0]
tree_sample_rate = trainset_speechcommands_tree[0][1]
```

```
show_waveform(cat_waveform, cat_sample_rate, 'cat')
```

```
Waveform: tensor([[-0.0028, -0.0054, -0.0034,  ..., -0.0011, -0.0013, -0.0014]])
Sample rate: 16000
Labels: yes

Shape of transformed waveform: torch.Size([1, 1600])
Sample rate: 1600.0
```



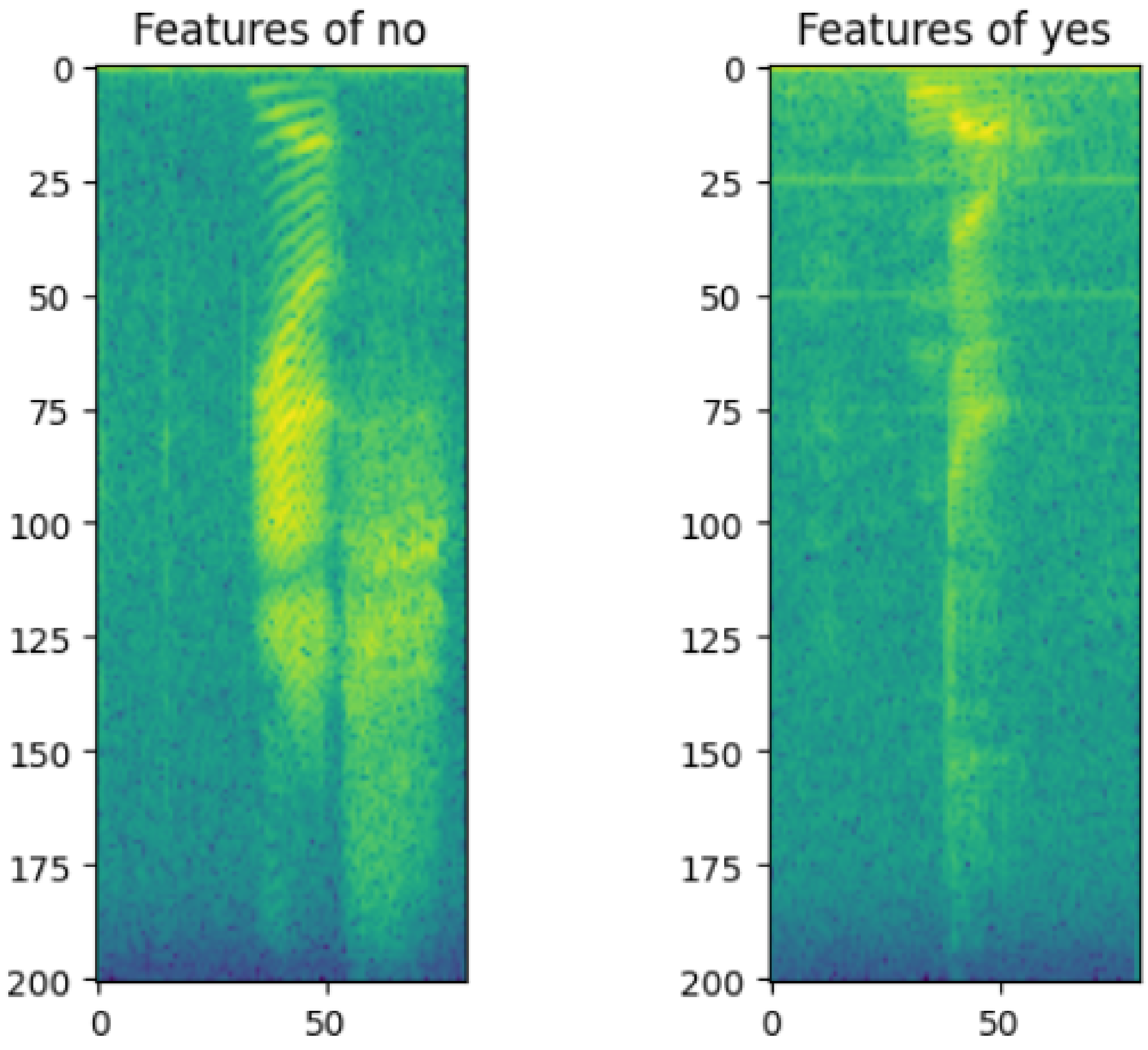
SPECTROGRAM

A spectrogram is a visual representation of the frequency content of a signal as it varies with time.

```
def show_spectrogram(waveform_classA, waveform_classB):
    yes_spectrogram = torchaudio.transforms.Spectrogram()(waveform_classA)
    print("\nShape of yes spectrogram: {}".format(yes_spectrogram.size()))
    no_spectrogram = torchaudio.transforms.Spectrogram()(waveform_classB)
    print("Shape of no spectrogram: {}".format(no_spectrogram.size()))
    plt.figure()
    plt.subplot(1, 2, 1)
    plt.title("Features of {}".format('no'))
    plt.imshow(yes_spectrogram.log2()[0,:,:].numpy(), cmap='viridis')
    plt.subplot(1, 2, 2)
    plt.title("Features of {}".format('yes'))
    plt.imshow(no_spectrogram.log2()[0,:,:].numpy(), cmap='viridis')
```

```
show_spectrogram(cat_waveform, tree_waveform)
```

```
Shape of yes spectrogram: torch.Size([1, 201, 81])
Shape of no spectrogram: torch.Size([1, 201, 81])
```



A Mel spectrogram is a specialized representation of the frequency content of an audio signal, designed to mimic human auditory perception. It is obtained by applying the Mel filterbank to the power spectrum of short, overlapping frames of the signal.

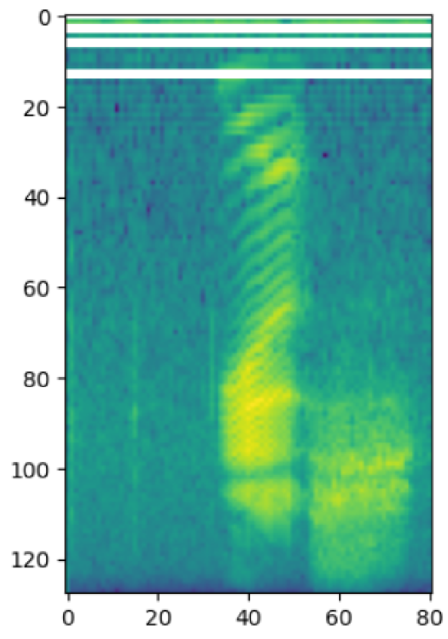
MELSPECTROGRAM

A Mel spectrogram is a specialized representation of the frequency content of an audio signal, designed to mimic human auditory perception. It is obtained by applying the Mel filterbank to the power spectrum of short, overlapping frames of the signal.

```
def show_melspectrogram(waveform,sample_rate):
    mel_spectrogram = torchaudio.transforms.MelSpectrogram(sample_rate)(waveform)
    print("Shape of spectrogram: {}".format(mel_spectrogram.size()))
    plt.figure()
    plt.imshow(mel_spectrogram.log2()[0,:,:].numpy(), cmap='viridis')
```

```
show_melspectrogram(cat_waveform, cat_sample_rate)
```

```
Shape of spectrogram: torch.Size([1, 128, 81])
```



Mel-Frequency Cepstral Coefficients

MFCCs are derived from the Mel spectrogram and represent the spectral characteristics of an audio signal in a way that is more perceptually relevant to human hearing.

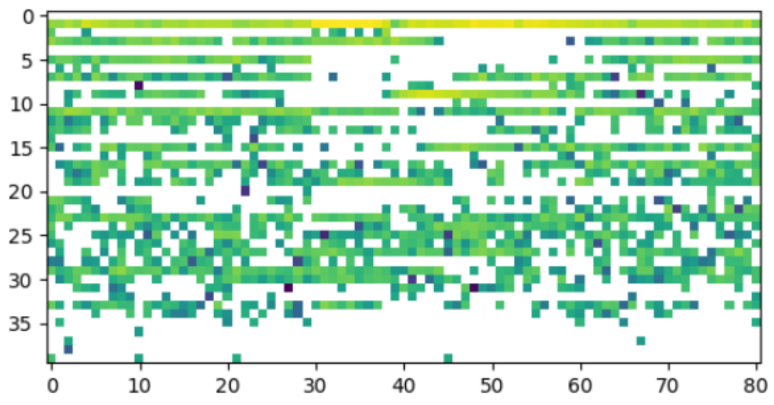
```
def show_mfcc(waveform,sample_rate):
    mfcc_spectrogram = torchaudio.transforms.MFCC(sample_rate= sample_rate)(waveform)
    print("Shape of spectrogram: {}".format(mfcc_spectrogram.size()))

    plt.figure()
    fig1 = plt.gcf()
    plt.imshow(mfcc_spectrogram.log2()[0,:,:].numpy(), cmap='viridis')

    plt.figure()
    plt.plot(mfcc_spectrogram.log2()[0,:,:].numpy())
    plt.draw()
```

```
show_mfcc(cat_waveform,  cat_sample_rate)
```

```
Shape of spectrogram: torch.Size([1, 40, 81])
```



CREATE SPECTROGRAM IMAGES IN IMAGEFOLDER

```
def create_spectrogram_images(trainloader, label_dir):
    directory = f'/kaggle/working/spectrograms/{label_dir}/'
    if(os.path.isdir(directory)):
        print("Data exists for", label_dir)
    else:
        os.makedirs(directory, mode=0o777, exist_ok=True)
        for i, data in enumerate(trainloader):
            waveform = data[0]
            sample_rate = data[1][0]
            label = data[2]
            ID = data[3]
            spectrogram_tensor = torchaudio.transforms.Spectrogram()(waveform)
            fig = plt.figure()
            plt.imsave(f'/kaggle/working/spectrograms/{label_dir}/spec_img{i}.png', spectrogram_tensor[0].log2()[
```

```
create_spectrogram_images(trainloader_yes, 'tree')
create_spectrogram_images(trainloader_no, 'cat')
```

```
Data exists for tree
Data exists for cat
```

```
data_path = '/kaggle/working/spectrograms' #looking in subfolder train

tree_cat_dataset = datasets.ImageFolder(
    root=data_path,
    transform=transforms.Compose([transforms.Resize((201,81)),
                                  transforms.ToTensor()
                                  ])
)
print(tree_cat_dataset)
```

```
Dataset ImageFolder
    Number of datapoints: 7985
    Root location: /kaggle/working/spectrograms
    StandardTransform
Transform: Compose(
               Resize(size=(201, 81), interpolation=bilinear, max_size=None, antialias=warn)
               ToTensor()
           )
```

```
class_map=tree_cat_dataset.class_to_idx
print("\nClass category and index of the images: {}\n".format(class_map))
```

```
Class category and index of the images: {'cat': 0, 'tree': 1}
```

```
#split data to test and train
#use 80% to train
train_size = int(0.8 * len(tree_cat_dataset))
test_size = len(tree_cat_dataset) - train_size
tree_cat_dataset, tree_cat_test_dataset = torch.utils.data.random_split(tree_cat_dataset, [train_size, test_size]

print("Training size:", len(tree_cat_dataset))
print("Testing size:",len(tree_cat_test_dataset))
```

```
Training size: 6388
Testing size: 1597
```

```
from collections import Counter
# labels in training set
train_classes = [label for _, label in tree_cat_dataset]
Counter(train_classes)
```

```
Counter({1: 3228, 0: 3160})
```

```
train_dataloader = torch.utils.data.DataLoader(
    tree_cat_dataset,
    batch_size=15,
    num_workers=2,
    shuffle=True
)
test_dataloader = torch.utils.data.DataLoader(
    tree_cat_test_dataset,
    batch_size=15,
    num_workers=2,
    shuffle=True
)
```

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

CNN

Spectrogram images capture the frequency content of the audio over time. The CNN trained on a dataset of these spectrogram images, learning to recognize patterns and features that correspond to different speech elements. Once trained, the model can take new spectrogram images and predict the corresponding speech transcription. This approach leverages the power of CNNs to automatically learn relevant features from the spectrogram data, contributing to effective speech recognition models.

```
class CNNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=5)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=5)
        self.conv2_drop = nn.Dropout2d()
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(51136, 50)
        self.fc2 = nn.Linear(50, 2)


    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        #x = x.view(x.size(0), -1)
        x = self.flatten(x)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = F.relu(self.fc2(x))
        return F.log_softmax(x,dim=1)

model = CNNet().to(device)
```

```python
# cost function used to determine best parameters
cost = torch.nn.CrossEntropyLoss()

learning_rate = 0.0001
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

# Create the training function
def train(dataloader, model, loss, optimizer):
    model.train()
    size = len(dataloader.dataset)
    for batch, (X, Y) in enumerate(dataloader):

        X, Y = X.to(device), Y.to(device)
        optimizer.zero_grad()
        pred = model(X)
        loss = cost(pred, Y)
        loss.backward()
        optimizer.step()

        if batch % 100 == 0:
            loss, current = loss.item(), batch * len(X)
            print(f'loss: {loss:>7f}  [{current:>5d}/{size:>5d}]')


# Create the validation/test function
def test(dataloader, model):
    size = len(dataloader.dataset)
    model.eval()
    test_loss, correct = 0, 0

    with torch.no_grad():
        for batch, (X, Y) in enumerate(dataloader):
            X, Y = X.to(device), Y.to(device)
            pred = model(X)

            test_loss += cost(pred, Y).item()
            correct += (pred.argmax(1)==Y).type(torch.float).sum().item()

    test_loss /= size
    correct /= size

    print(f'\nTest Error:\nacc: {(100*correct):>0.1f}%, avg loss: {test_loss:>8f}\n')
```

TRAIN

```python
epochs = 15
for t in range(epochs):
    print(f'Epoch {t+1}\n-------------------------------')
    train(train_dataloader, model, cost, optimizer)
    test(test_dataloader, model)
print('Done!')
```

```
Epoch 1
-------------------------------
loss: 0.691589  [    0/ 6388]
loss: 0.598309  [ 1500/ 6388]
loss: 0.447526  [ 3000/ 6388]
loss: 0.327089  [ 4500/ 6388]
loss: 0.514054  [ 6000/ 6388]

Test Error:
acc: 90.4%, avg loss: 0.014237
```

TEST

```
model.eval()
test_loss, correct = 0, 0
class_map = ['cat', 'tree']
correct=0
total = 0
with torch.no_grad():
    for batch, (X, Y) in enumerate(test_dataloader):
        total+=1
        X, Y = X.to(device), Y.to(device)
        pred = model(X)
        if Y[0] == pred[0].argmax(0):
            correct+=1
print(correct/total)
```

```
0.9345794392523364
```

MERCI