Natural Language Processing Speech Recognition 1

Fall, 2024 (12^{th} week)

School of AI Convergence

Jangmin Oh

Overview

- https://huggingface.co/learn/audio-course
- Working with Audio Data
- Audio Applications
- Transformer Architectures

Nature of Audio Data

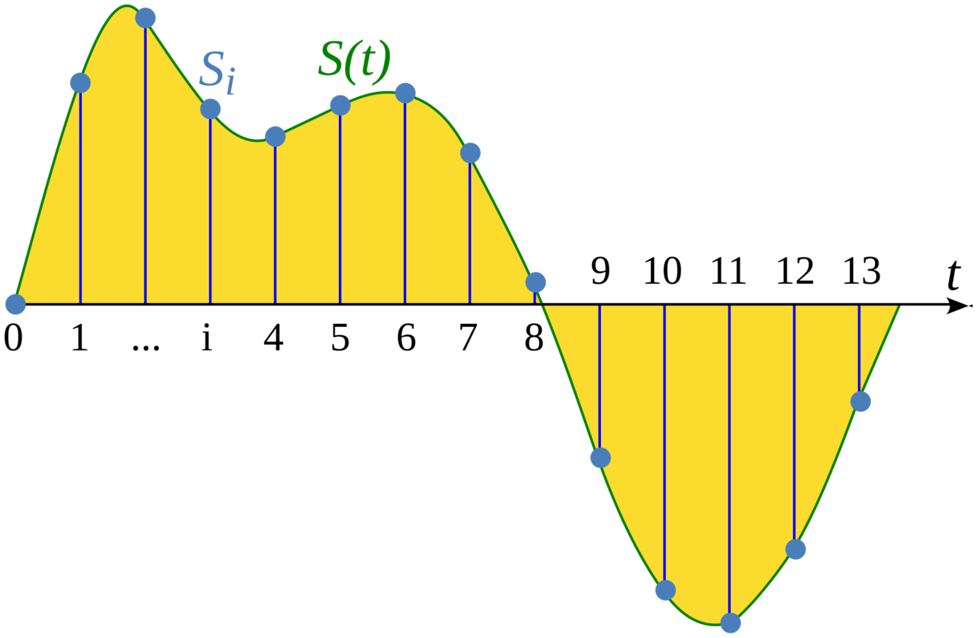
- Key Points:
  - Sound waves are continuous signals with infinite values in time
  - Digital devices require finite arrays - converting continuous signals into digital representation
- Digital Audio Formats
  - Common formats: .wav, .flac, .mp3
  - Differences: Compression methods and efficiency
- From Analog to Digital
  - Microphone: Captures sound waves -> Electrical signal
  - Analog-to-Digital Converter (ADC): Converts electrical signal to digital via sampling

Sampling and Sampling Rate

- **Sampling**: Measures the signal at fixed time intervals.
- **Sampling Rate**:
  - Defines the number of samples per second (Hz).
  - Examples:
    - CD-quality: 44,100 Hz.
    - Speech models: 16,000 Hz (sufficient for human speech).
- **Nyquist Limit**: Highest frequency = Half the sampling rate.
- **Note**: Low rates (e.g., 8,000 Hz) lead to muffled speech.

Sampling and Sampling Rate

- **Sampled Wave** (discrete)



Importance of Consistent Sampling Rates

- **Why it matters**:
  - Consistency ensures better generalization in models.
  - Resampling aligns sampling rates during preprocessing.
- **Example**:
  - 5-second audio:
    - At 16 kHz → 80,000 samples.
    - At 8 kHz → 40,000 samples.

Amplitude and Bit Depth

- **Amplitude**:
  - Represents sound pressure (loudness).
  - Measured in decibels (dB).
- **Bit Depth**:
  - Precision of amplitude values:
    - 16-bit: 65,536 steps.
    - 24-bit: 16,777,216 steps.
- Higher bit depth = Lower quantization noise.
- For Machine Learning
  - float32 (24 bit precision)
- Digital Audio Signal: 0dB (most loud): every -6dB halves the amplitude. (-60dB is inaudible)

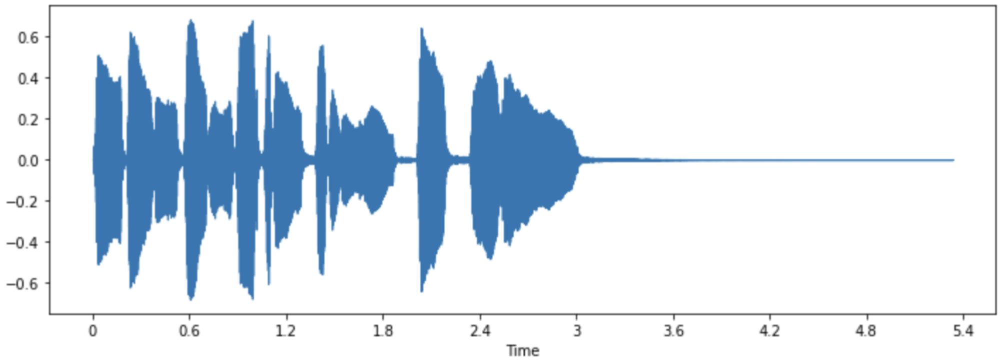
Waveform Representation

- **Definition**: Plots sample values over time (time-domain representation).
- **Use Cases**:
  - Identify features like timing, loudness, noise.
  - Debug preprocessing or model errors.
- **Python Example**:
  - Code snippet using librosa to load and visualize a waveform.

librosa

```
import librosa

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

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

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

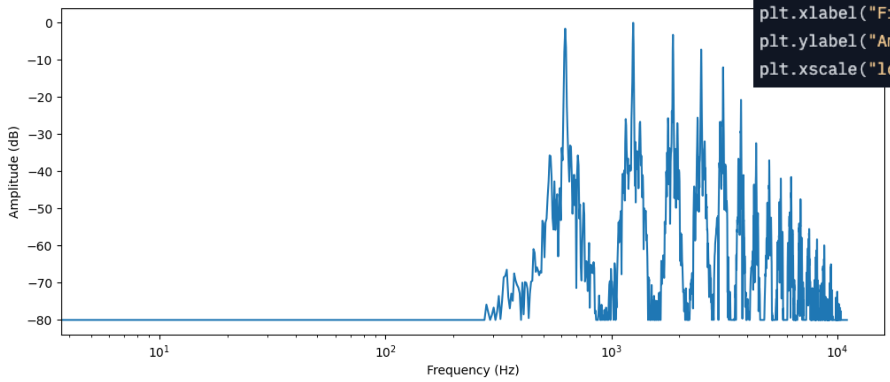


Frequency Spectrum

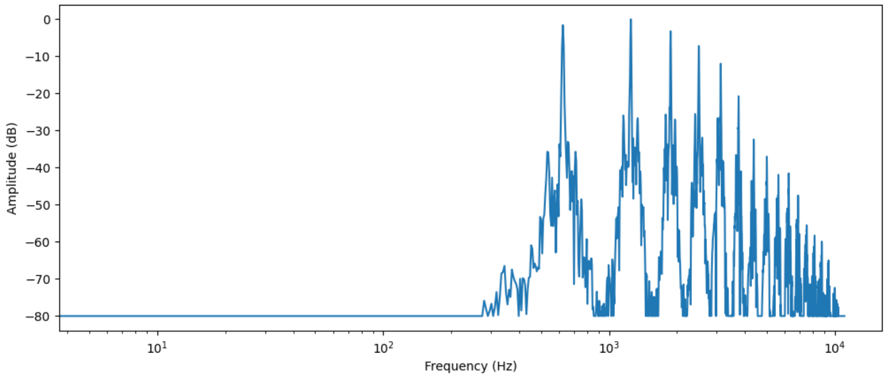
- **Definition**: Visualizes frequency components of a signal.
- **Key Concepts**:
  - Uses Discrete Fourier Transform (DFT).
  - Amplitude spectrum in decibels (dB).
- **Python Example**:
  - Code snippet using numpy and librosa to compute and plot the spectrum.

- First 4096 samples (the length of the first note being played)

```python
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")
```



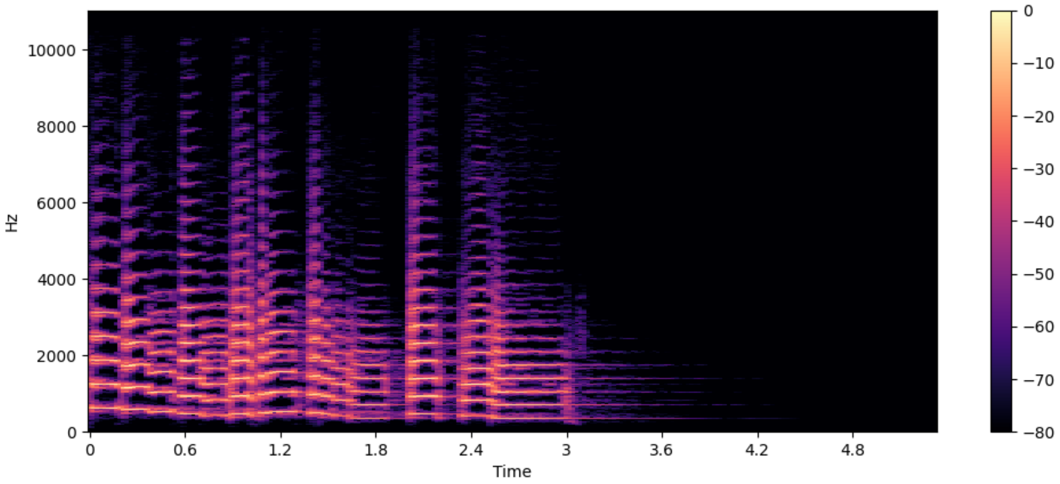
- x-axis: frequency values in a logarithmic scale
- y-axis: amplitudes at the frequency
- peaks: harmonics of the note that's being playes
  - a trumpet generates a fundamental frequency (first harmonic E^b: 620Hz) + multiples of the frequency (2x620Hz, 3x620Hz, …)



Spectrograms

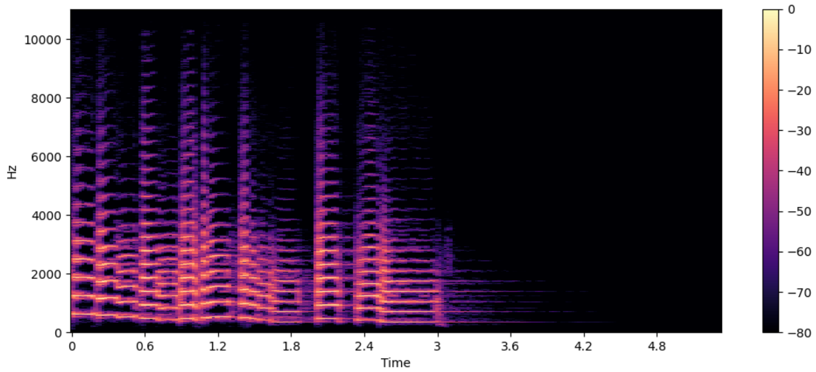
- **The Problem of Spectrum:** only shows a frozen snapshot of the frequencies at a given instant
- **Spectrograms!!**
- **Definition**: Shows how frequencies change over time (time-frequency representation).
- **Creation**:
  - Short Time Fourier Transform (STFT).
  - Vertical slices represent individual frequency spectra.
- **Python Example**:
  - Code snippet using librosa.stft() and specshow().

```
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()
```



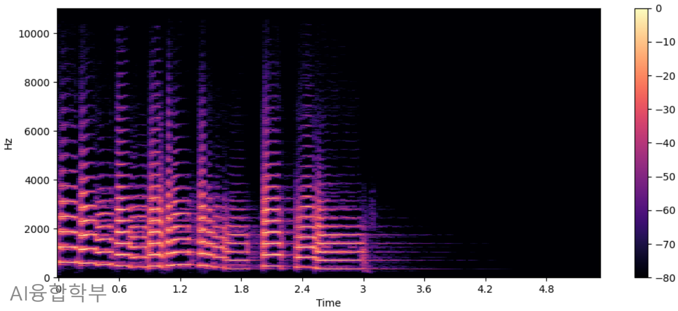
Spectrogram

- **x-axis:** Represents time (in seconds), similar to waveform visualizations.
- **y-axis:** Represents frequency (in Hz).
- **Color Intensity:** Indicates the amplitude or power of each frequency component at a given time, measured in decibels (dB).



How the Spectrogram is Created

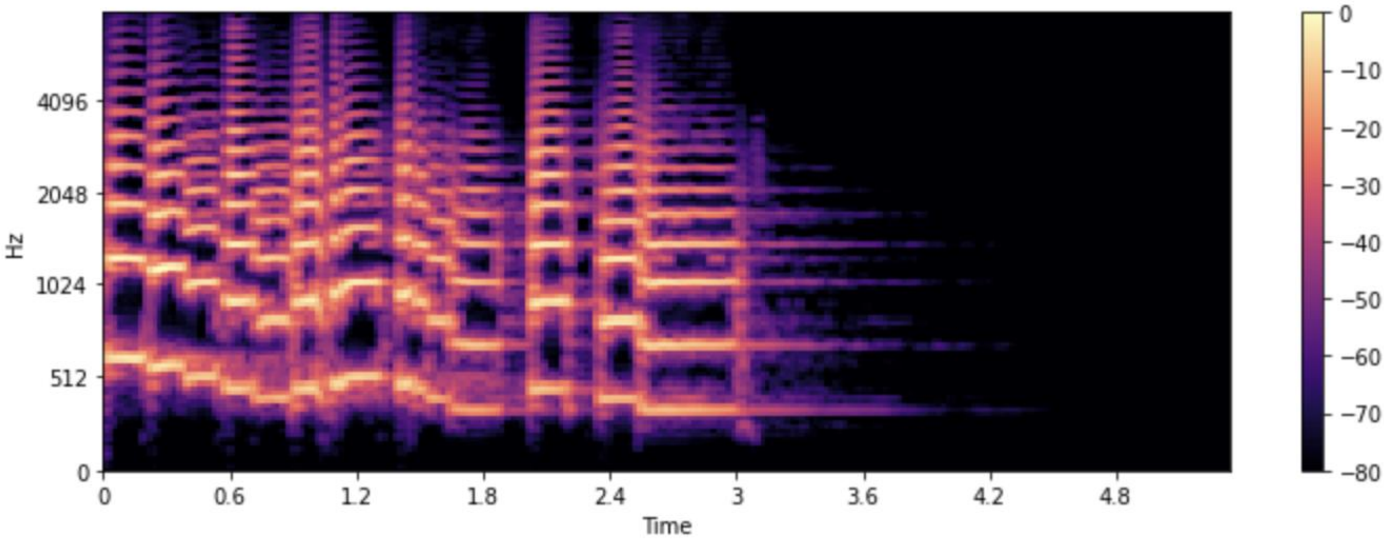
1. **Segmenting the Audio Signal:** The audio signal is divided into short segments, each lasting a few milliseconds.
2. **Calculating Frequency Spectrum:** For each segment, the **Discrete Fourier Transform (DFT)** is applied to calculate the frequency spectrum.
3. **Stacking Spectra:** The resulting spectra are aligned along the time axis, creating a **2D representation** of time vs. frequency.
4. **Vertical Slices:** Each vertical slice represents the frequency spectrum for a single segment of the audio.



Mel Spectrogram

- **Definition**: Frequency axis adjusted to match human hearing (mel scale).
  - Human auditory system is more sensitive to changes in lower frequencies. (The sensitivity decreases logarithmically.)
- **Advantages**:
  - Captures perceptually meaningful features.
  - Widely used in speech tasks like recognition, identification.
- **Python Example**:
  - Code snippet using `librosa.feature.melspectrogram()`.

```python
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()
```



```python
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()
```

- **Key parameters**
- **n_mels:** Defines the number of mel bands to generate.
  - **Mel Bands**: Divide the frequency spectrum into perceptually meaningful components, using filters that mimic human ear sensitivity to frequencies.
  - **Typical Values**: Common values for n_mels are 40 or 80.
- **fmax:** Sets the maximum frequency (in Hz) that we care about in the analysis.

dataset for Audio

- Lots of dataset in the HuggingFace's hub
- An Example: MINDS-14
  - recordings of people asking an e-banking system questions

```python
from datasets import load_dataset

minds = load_dataset("PolyAI/minds14", name="en-AU", split="train")
minds
```

```
Dataset(
    {
        features: [
            "path",
            "audio",
            "transcription",
            "english_transcription",
            "intent_class",
            "lang_id",
        ],
        num_rows: 654,
    }
)
```

dataset for Audio

- a sample from MINDS-14

```
example = minds[0]
example
```

```
{
    "path": "/root/.cache/huggingface/datasets/downloads/extracte
    "audio": {
        "path": "/root/.cache/huggingface/datasets/downloads/extr
        "array": array(
            [0.0, 0.00024414, -0.00024414, ..., -0.00024414, 0.00
            dtype=float32,
        ),
        "sampling_rate": 8000,
    },
    "transcription": "I would like to pay my electricity bill usi
    "english_transcription": "I would like to pay my electricity
    "intent_class": 13,
    "lang_id": 2,
}
```

```
Dataset(
    {
        features: [
            "path",
            "audio",
            "transcription",
            "english_transcription",
            "intent_class",
            "lang_id",
        ],
        num_rows: 654,
    }
)
```

dataset for Audio

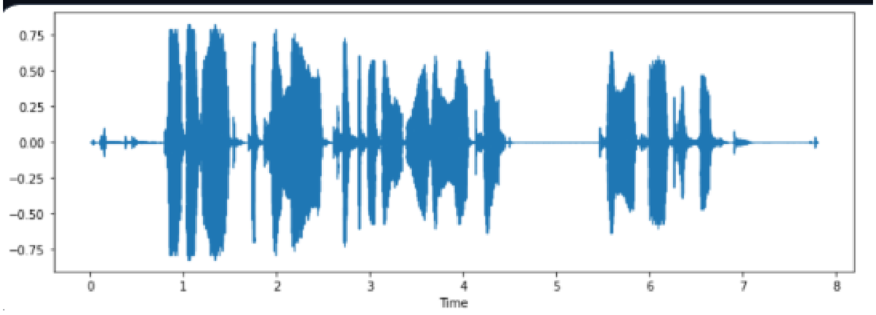
- Play the audio

```
import gradio as gr


def generate_audio():
    example = minds.shuffle()[0]
    audio = example["audio"]
    return (
        audio["sampling_rate"],
        audio["array"],
    ), id2label(example["intent_class"])


with gr.Blocks() as demo:
    with gr.Column():
        for _ in range(4):
            audio, label = generate_audio()
            output = gr.Audio(audio, label=label)

demo.launch(debug=True)
```

dataset for Audio

- plot the waveform

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

array = example["audio"]["array"]
sampling_rate = example["audio"]["sampling_rate"]

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



Preprocessing an audio dataset

- Need for preprocessing
  - to train and to inference
- Procedures
  - Resampling the audio data
  - Filtering the dataset
  - Converting audio data to model's expected input

Resampling the audio data

- load_dataset: loads examples with the specified sampling rate
- Most of the pretrained models were pretrained on datasets (sampled at 16kHz)
- MINDS-14 dataset (sampled at 8kHz): Needs to upsample.
- cast_column methods

```python
from datasets import Audio

minds = minds.cast_column("audio", Audio(sampling_rate=16_000))
```

Resampling the audio data

- upsampled version of sample.
- NOTE
  - Resampling is tricky…
  - (Nyquist sampling theorem)
    - if sampled at 8kHZ, the audio does not contain any frequency over 4kHz
  - If we downsample from 16kHz into 8kHz, just throwing every other sample is not sufficient.
  - Use package librosa or HuggingFace's datasets


```
{
    "path": "/root/.cache/huggingface/datasets/downloads/extracte
    "audio": {
        "path": "/root/.cache/huggingface/datasets/downloads/extr
        "array": array(
            [
                2.0634243e-05,
                1.9437837e-04,
                2.2419340e-04,
                ...,
                9.3852862e-04,
                1.1302452e-03,
                7.1531429e-04,
            ],
            dtype=float32,
        ),
        "sampling_rate": 16000,
    },
    "transcription": "I would like to pay my electricity bill usi
    "intent_class": 13,
}
```

Filtering the dataset

- Use dataset's filter method
- i.e) If we want to keep samples longer than 20s

```python
MAX_DURATION_IN_SECONDS = 20.0


def is_audio_length_in_range(input_length):
    return input_length < MAX_DURATION_IN_SECONDS
```

```python
# use librosa to get example's duration from the audio file
new_column = [librosa.get_duration(path=x) for x in minds["path"]]
minds = minds.add_column("duration", new_column)

# use 🤗 Datasets' `filter` method to apply the filtering function
minds = minds.filter(is_audio_length_in_range, input_columns=["duration"])

# remove the temporary helper column
minds = minds.remove_columns(["duration"])
minds
```

Preprocessing audio data

Key Challenges:

1. **Raw Audio Complexity**:
   - Raw audio data consists of continuous sample values.
   - Requires transformation into model-compatible features.
2. **Model-Specific Requirements**:
   - Input features depend on the model's architecture and pre-training data.
   - Example: Whisper uses log-mel spectrograms, others may use raw waveforms.
3. **Standardization**:
   - Features must be consistent across datasets for reliable performance.

Solution:

- HuggingFace's transformers provide feature extractors for supported models

An example of feature extractor

- **Whisper**:
  - Pre-trained model for Automatic Speech Recognition (ASR)
  - Published by OpenAI (Sept 2022, we'll cover later)
- **Key Transformations**:
  - **Padding/Truncation**:
    - Standardizes audio length to 30 seconds.
    - Shorter examples padded with zero.
    - longer truncated.
    - No attention mask (all examples have an input length of 30s).
  - **Log-Mel Spectrograms**:
    - Represents frequency changes over time on the mel scale.
    - Expressed in decibels for human-hearing alignment.
    - Creates perceptually meaningful input features for models.

Whisper's Feature Extractor

- Loading

```python
from transformers import WhisperFeatureExtractor

feature_extractor = WhisperFeatureExtractor.from_pretrained("openai/whisper-small")
```

- Preprocessing

```python
def prepare_dataset(example):
    audio = example["audio"]
    features = feature_extractor(
        audio["array"], sampling_rate=audio["sampling_rate"], padding=True
    )
    return features
```

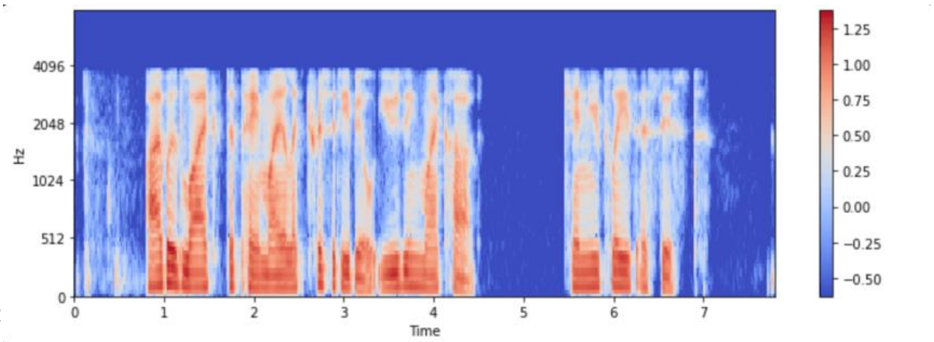
```python
minds = minds.map(prepare_dataset)
minds
```

Whisper's Feature Extractor

- Preprocessed Dataset

```
Dataset(
    {
        features: ["path", "audio", "transcription", "intent_class", "input_features"],
        num_rows: 624,
    }
)
```

  - input_features: log-mel spectrograms



Whisper's Feature Extractor

- config
  - n_mels (feature_size): 80
  - hop_length: 160
  - chunk_length: 30
  - sampling_rate: 16000
- Processed Tensor shape
  - (1, 80, 3000)
  - 80: number of mel filters (set to n_mels)
  - 3000: 16000 / 160 x 30
    - 16000: samples per second
    - 160: STFT is performed at 160 sample interval (10ms)
    - 16000/160 = 100 STFTs per second (frames)
    - 16000/160 x30 = 3000 frames

Processor for Multimodal Models

- Processor: combines feature extractor and tokenizer

```
from transformers import AutoProcessor

processor = AutoProcessor.from_pretrained("openai/whisper-small")
```

- Use case
  - simplifies handling multimodal tasks like ASR

Audio Applications

- Audio classification
- Automatic speech recognition (ASR)
- Speaker diarization
- Text to Speech (TTS)

Audio classification with a pipeline

- classify the intent of MINDS-14 dataset's recordings
  - intent_class: target label
- Upsample

```
from datasets import load_dataset
from datasets import Audio

minds = load_dataset("PolyAI/minds14", name="en-AU", split="train")
minds = minds.cast_column("audio", Audio(sampling_rate=16_000))
```

```
example = minds[0]
```

```
classifier(example["audio"]["array"])
```

- Use a pipeline with a pretrained model

```
from transformers import pipeline

classifier = pipeline(
    "audio-classification",
    model="anton-l/xtreme_s_xlsr_300m_minds14",
)
```

```
[
    {"score": 0.9631525278091431, "label": "pay_bill"},
    {"score": 0.02819698303937912, "label": "freeze"},
    {"score": 0.0032787492964416742, "label": "card_issues"},
    {"score": 0.0019414445850998163, "label": "abroad"},
    {"score": 0.0008378693601116538, "label": "high_value_payment"}
]
```

ASR wit a pipeline

```python
from transformers import pipeline

asr = pipeline("automatic-speech-recognition")
```

```python
example = minds[0]
asr(example["audio"]["array"])
```

Output:

```
{"text": "I WOULD LIKE TO PAY MY ELECTRICITY BILL USING MY COD CAN YOU PLEASE ASSIST"}
```

```python
example["english_transcription"]
```

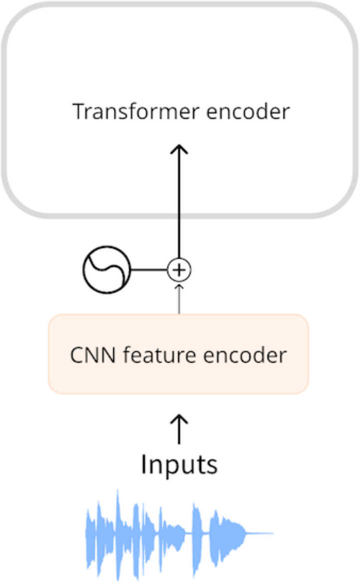
```
"I would like to pay my electricity bill using my card can you please assist"
```

Transformer Architecture for Audio: Input

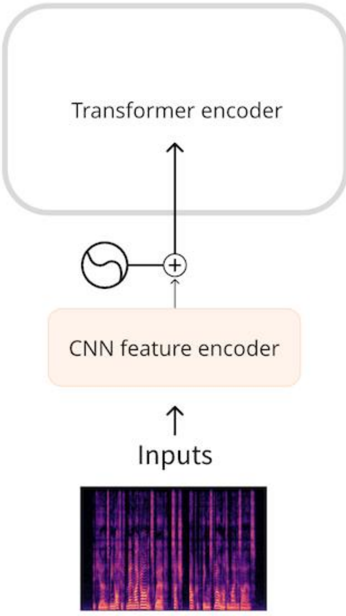
Models
- Wav2Vec2
- HuBERT

sequence size
- 30x16k = 380k

waveform input



spectrogram input

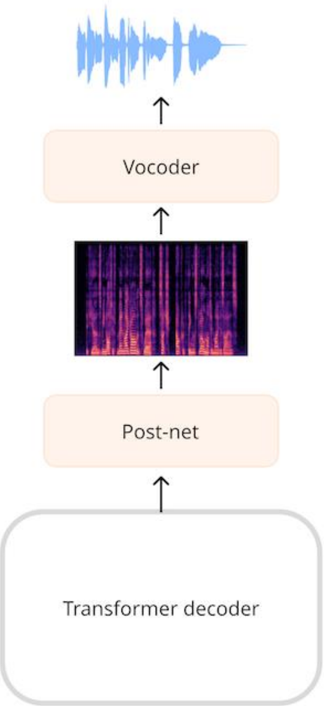


Models
- Whisper

sample size
- (80, 3000)

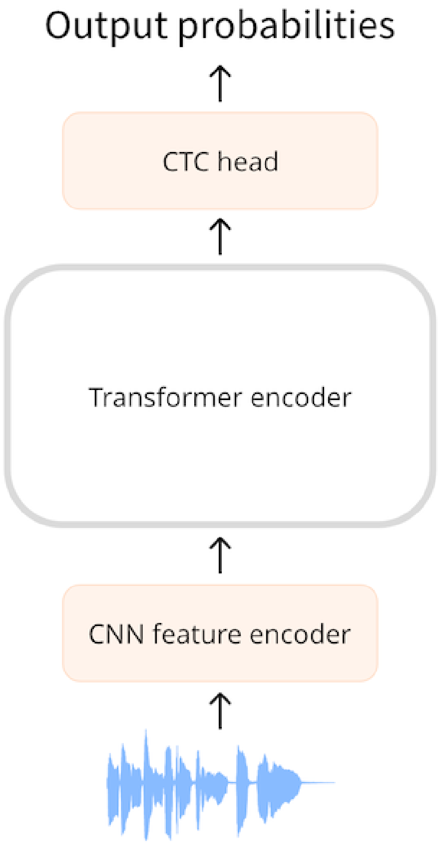
Transformer Architecture for Audio: Output

- Text output (ASR)
- Spectrogram output (TTS)
  - Additional network (VOCODER)
    - convert the spectrogram into a waveform
  - STFT <-> ISTFT
    - requires both amplitude and phase information
    - But phase info was not modeled in most audio models
    - VOCODER estimate the phase information to convert spectrogram into a waveform



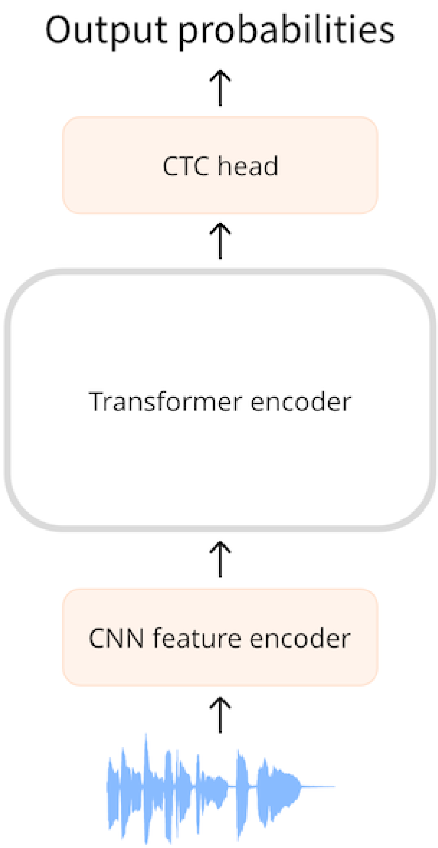
CTC architectures

- **Definition**: Connectionist Temporal Classification (CTC)
  - is a method used with **encoder-only transformers** for **automatic speech recognition (ASR)**.
- **Examples**:
  - **Wav2Vec2**: Processes raw audio waveforms.
  - **HuBERT**: Predicts discrete speech units.
  - **M-CTC-T**: Designed for multilingual ASR (e.g., includes Chinese characters).
- **Key Idea**:
  - Encodes audio into hidden-states.
  - Adds a small classification head to predict characters.



CTC architectures

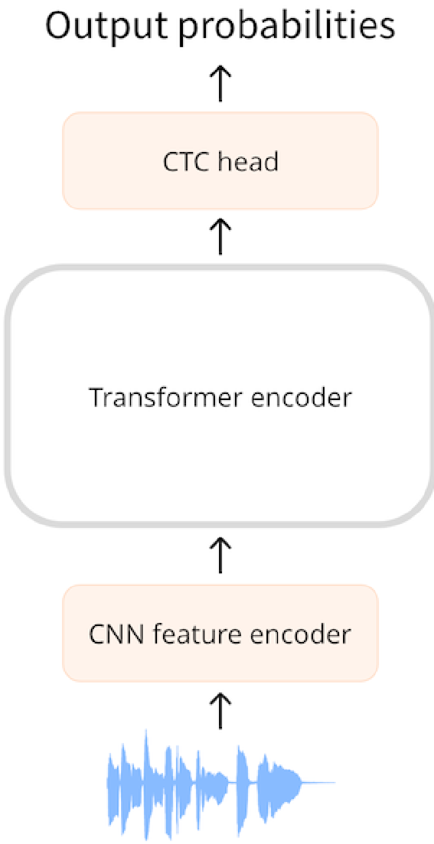
- **Encoder-Only Transformer with CTC**
- **Architecture**:
  - **Encoder**: Maps audio waveform → hidden-states.
  - **CTC Head**: Linear mapping to character labels.
- **Challenges**:
  - No timing alignment between audio and text.
  - CTC uses a **blank token** to handle duplicate predictions.
- **CTC Loss**:
  - Matches predicted sequences to transcriptions without explicit alignment.



CTC architectures

- **How CTC Handles Alignment**
- **The Problem**:
  - Audio has more frames than transcription characters.
  - Predictions include duplicates (e.g., "ERRRRORR").
- **CTC Solution**:
  - **Blank Token (_)** separates repeated characters.
  - Removes duplicates within groups.
  - Eliminates blank tokens for final output:
  - _ER_RRR_ORR -> _RER_R_OR -> ERROR

- **Output**:
  - Predictions align without destroying word structure.



CTC architectures

Example Workflow

1. Input:

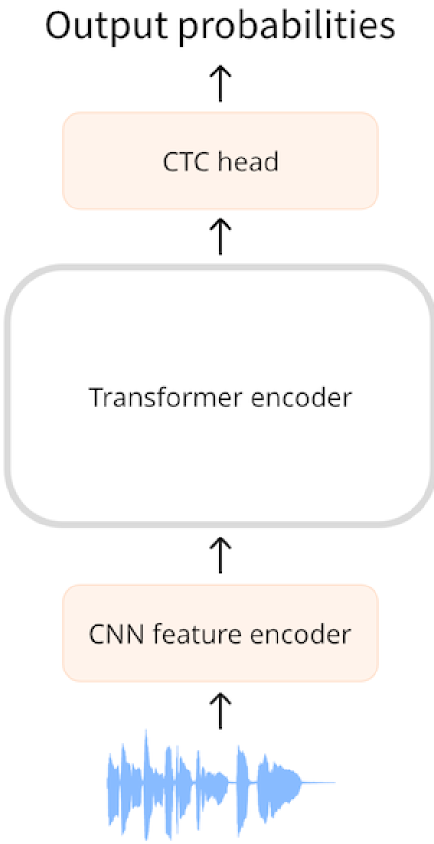
- 1-second audio file → 50 hidden-states (20ms each).
- Encoder output: Shape = (768,50).

2. CTC Head:

- Maps hidden-states to logits: (50,32).
- Vocabulary includes characters, blank token, and separators.

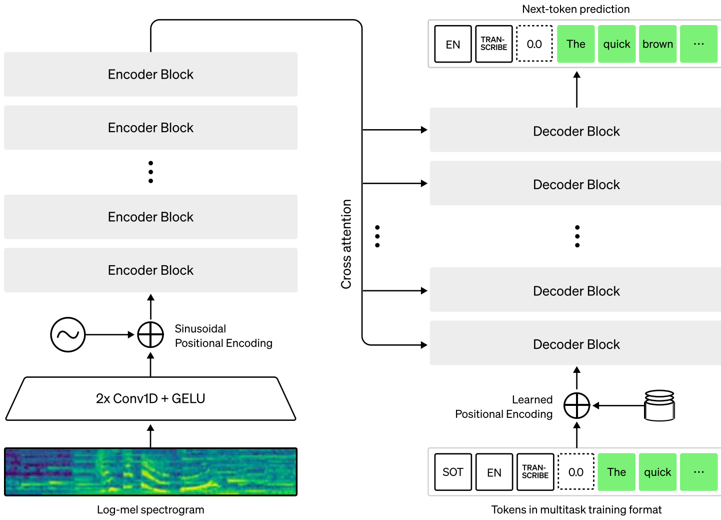
3. Decoding:

- Combines predictions into final transcription.



Seq2Seq architectures

- **Definition**:
  - Sequence-to-sequence (seq2seq) models
  - map an input sequence to an output sequence of possibly different lengths.
- **Architecture**:
  - Combines **encoder** and **decoder** components of a transformer.
  - Encoder extracts features from input, while decoder generates outputs autoregressively.
- **Applications**:
  - **NLP**: Translation, summarization.
  - **Audio**: Automatic Speech Recognition (ASR), Text-to-Speech (TTS).
- **Difference from CTC**:
  - No one-to-one correspondence between input and output sequences.

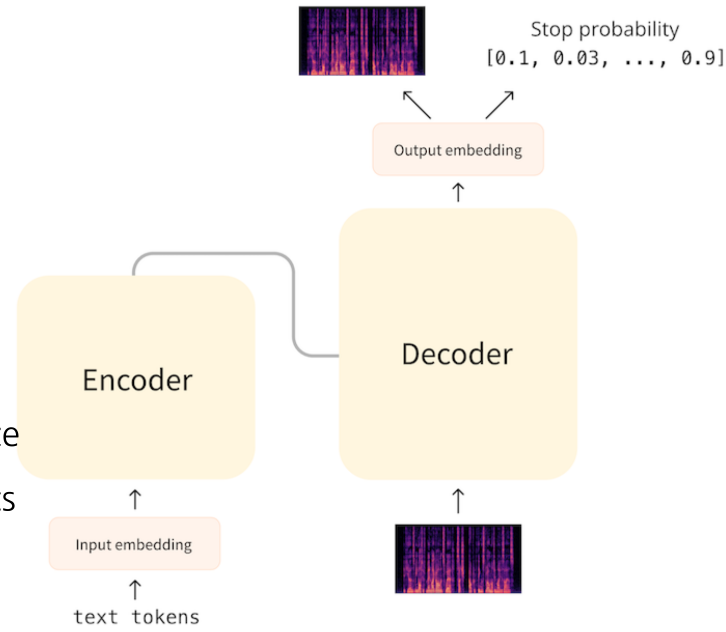


Seq2Seq architectures

- **ASR with Seq2Seq Models**
- **Whisper Architecture**:
  - **Encoder**: Processes log-mel spectrograms to generate hidden states.
  - **Decoder**:
    - Uses cross-attention to generate text tokens.
    - Operates autoregressively, predicting one token at a time.
- **Key Features**:
  - **Cross-Attention**: Links encoder outputs to decoder inputs.
  - **Causal Attention**: Prevents decoder from seeing future tokens.
- **Advantages**:
  - Outputs full words or portions (GPT-2 tokenizer with 50k+ tokens).
  - End-to-end training provides flexibility and better performance.

Seq2Seq architectures

- **Seq2Seq in TTS**
- **Process**:
  - **Encoder**: Maps text tokens → hidden states.
  - **Decoder**: Generates spectrograms one timestep at a time.
  - **Post-Net**: Refines generated spectrograms with convolutional layers.
- **Stopping Criterion**:
  - Decoder predicts when to stop based on a probability threshold.
- **Challenges**:
  - One-to-many mapping: Multiple ways to vocalize the same text.
  - Evaluation: Requires human listeners and metrics like MOS (Mean Opinion Score).



CTC vs Seq2Seq

Feature	**Seq2Seq**	**CTC**
Architecture	Encoder-Decoder	Encoder-Only
Output Tokens	Full words or subwords	Characters
Alignment	Implicit through cross-attention	Explicit via blank tokens
Training	End-to-end with cross-entropy loss	CTC loss
Performance	Superior transcription quality	Simpler, faster decoding
Applications	ASR, TTS, Translation	Primarily ASR

Summary

- Introduction to Audio Data
- (log-)mel spectrogram
- Audio Applications
- Transformer Architectures
- Next Week (week-13)
  - Whisper Model
  - Fine Tuning and Inference