TorchAudio Tutorial at ISCSLP 2022

Xiaohui Zhang, Zhaoheng Ni, Jeff Hwang, Caroline Chen

PyTorch, Meta.
OUR TEAM

- Moto Hira, Software Engineer
- Caroline Chen, Software Engineer
- Jeff Hwang, Software Engineer
- Zhaoheng Ni, Research Scientist
- Xiaohui Zhang, Research Scientist
- Yumeng Tao, Engineering Manager
OUR MISSION

Accelerate research and productization of audio/speech ML, by delivering intuitive, well-documented, and performant audio/speech-oriented PyTorch components and APIs.
Agenda

01 Overview of TorchAudio
02 Source Separation and Speech Enhancement with TorchAudio
03 Streaming I/O and ASR with TorchAudio
04 Self-Supervised Learning (SSL) with TorchAudio
01 Overview of TorchAudio

Xiaohui Zhang
Research Scientist
Meta Platforms, Inc.
In this section, we will cover

1.1 What makes audio ML challenging for PyTorch users?
1.2 What is TorchAudio?
1.3 How do we operate TorchAudio?
1.1

WHAT MAKES AUDIO ML CHALLENGING FOR PYTORCH USERS?
WHAT MAKES AUDIO ML CHALLENGING FOR PYTORCH USERS?

• Audio data is complex
  ○ time series
  ○ streaming processing
WHAT MAKES AUDIO ML CHALLENGING FOR PYTORCH USERS?

• Audio data is complex
  ○ time series
  ○ streaming processing

• Audio ML tasks are deeply interdisciplinary
  ○ signal processing, NLP
  ○ multi-modalities
WHAT MAKES AUDIO ML CHALLENGING FOR PYTORCH USERS?

• Audio data is complex
  ○ time series
  ○ streaming processing

• Audio ML tasks are deeply interdisciplinary
  ○ signal processing, NLP
  ○ multi-modalities

• PyTorch (core library) lacks support for audio
  ○ signal processing
  ▪ streaming data I/O and decision making
1.2
WHAT IS TORCHAUDIO?
WHAT IS TORCHAUDIO?

To meet the aforementioned challenges, TorchAudio was developed. TorchAudio is an open-source library that extends PyTorch to the audio domain, serving two main purposes:

- Compensate for the lack of audio support in PyTorch
  → Make PyTorch-based audio ML practitioners’ life easier
- Lower the high barrier of entry to audio/speech ML from research through production
  → Attract more talents to the audio ML community
WHAT IS TORCHAUDIO?

Specifically, TorchAudio is a library of easy-to-use, well-documented, and performant audio/speech-oriented PyTorch components and APIs covering:

**Audio data utilities**
- Datasets
- Data I/O: StreamReader/Writer

**Audio-specific data transforms**
- Feature extraction
- Data augmentation
- Signal processing/synthesis

**Audio ML model architectures and pre-trained checkpoints,**
- ASR: Emformer, Conformer…
- Self-Supervised Learning (SSL): HuBERT, Wav2Vec2…
- Source Separation: Hybrid Demux
- TTS: WaveRNN, Tacotron 2

**Audio ML training loss functions & decoder**
- RNN-T loss
- CTC decoder
WHAT IS TORCHAUDIO NOT?

• TorchAudio serves **fundamental building blocks** for audio ML frameworks/ecosystems
  ○ *yet TorchAudio is not a training framework or closed ecosystem.*

• TorchAudio holds **end-to-end example recipes** to demonstrate its APIs and components
  ○ *yet TorchAudio is not a training recipe holder.*

• TorchAudio holds **popular pre-trained models** to facilitate their adoption by the community
  ○ *yet TorchAudio is not a model zoo.*
WHAT IS TORCHAUDIO?

[Key Characteristics]

• Easy-to-use
  o PyTorch is the only mandatory requirement
  o Available via pip, conda
  o Preinstalled on Google Colab

• Well-documented
  o Thorough documentation
  o Intuitive tutorials

• Performant
  o TorchAudio APIs implement primitive PyTorch interfaces and core PyTorch features, e.g. GPU acceleration, TorchScript, AutoGrad
WHAT IS TORCHAUDIO?

[Key Characteristics]

- Easy-to-use
  - PyTorch is the only mandatory requirement
  - Available via pip, conda
  - Preinstalled on Google Colab
- Well-documented
  - Thorough documentation
  - Intuitive tutorials
- Performant
  - TorchAudio APIs implement primitive PyTorch interfaces and core PyTorch features, e.g. GPU acceleration, TorchScript, AutoGrad

Therefore, TorchAudio components can:

- integrate easily with PyTorch-based audio ML projects
- facilitate development and reduce maintenance costs for major speech/audio ML toolkits, like SpeechBrain, ESPNet, HuggingFace, and K2.
WHAT IS TORCHAUDIO?

- Major open source projects that use Torchaudio
WHAT IS TORCHAUDIO?

- Major open source projects that use Torchaudio

Examples:
- Torchaudio’s RNN-T loss has been adopted by SpeechBrain and K2.
- Torchaudio’s Emformer model has been adopted by HuggingFace and K2
- Torchaudio’s data I/O, feature extraction and augmentation utilities has been very extensively adopted across both academia and industry.
import torchaudio

# Load audio data
waveform, sample_rate = torchaudio.load('original.flac')

# Resample to 8000 Hz
new_sample_rate = 8000
waveform = torchaudio.functional.resample(waveform, sample_rate, new_sample_rate)

# Save the audio
torchaudio.save('resampled.flac', waveform, new_sample_rate)
WHAT IS TORCH AUDIO? — A QUICK LIBRARY WALKTHROUGH

Datasets

```python
import torch
import torchaudio

dataset = torchaudio.datasets.LIBRISPEECH('path/to/dataset/dir', ...
)  
data_loader = torch.utils.data.DataLoader(dataset, ...)

for batch in data_loader:
    # operate on each batch of data
```
Feature extraction and augmentation

```python
import torchaudio.transforms as T

# Get spectrogram
trans = T.Spectrogram(...)
spectrogram = trans(waveform)

# Mask along time axis a.k.a SpecAugment
time_masking = T.TimeMasking(...)
time_masked = time_masking(spectrogram)
```
WHAT IS TORCH AUDIO? — A QUICK LIBRARY WALKTHROUGH

**Pipelines**

```python
import torchaudio

# pretrained model identifier
bundle = torchaudio.pipelines.EMFORMER_RNNT_BASE_LIBRISPEECH

# a model instance with pre-trained weights (Emformer RNN-T)
decoder = bundle.get_decoder()

# a preprocessor corresponding to audio processing steps for the Emformer RNN-T model.
feature_extractor = bundle.get_streaming_feature_extractor()

# a post-processor mapping decoded tokens to transcripts
token_processor = bundle.get_token_processor()

# apply the pipeline on waveform to get transcripts
features, length = feature_extractor(waveform)

hypos, state = decoder.infer(features, length, 10, state=state, hypothesis=hypothesis)

transcript = token_processor(hypothesis.tokens, lstrip=False)
```
1.3

HOW DO WE OPERATE TORCHAUDIO?
HOW DO WE OPERATE TORCHAUDIO?

As a **small team** serving a **huge community**, we heavily rely on external support:

- **Internal collaborations**: focusing on open sourcing SOTA modeling techniques
- **Community support**: bug fixes, performance improvement, feature proposal
- **Advisory board**
  - Keep us informed of the trends in the audio industry and provide directional guidance
  - Help us identify market opportunities, potential collaborators, contributors, and customers.
HOW DO WE OPERATE TORCHAUDIO?

Advisory Board Members:

- Abdelrahman Mohamed, Research Scientist, Meta
- Mei-Yuh Hwang, Research Scientist, Meta, IEEE Fellow
- Thomas Lunner, Director, Meta
- Xin Lei, Research Scientist Manager, Meta
- Daniel Povey, Chief Speech Scientist, Xiaomi Inc., IEEE Fellow
- Hung-Yi Lee, Assistant Professor, National Taiwan University
- Mirco Ravanelli, Associate Professor, Concordia University
- Shinji Watanabe, Associate Professor, Carnegie Mellon University, IEEE Fellow
HOW DO WE OPERATE TORCHAUDIO?

Key Technical Pillars

Pillars where we are actively investing:

**Data utilities**
- Datasets and data I/O
- Feature extraction and data augmentation
- Data alignment and segmentation

**Self-supervised Learning (SSL)**
- Model bundles and pre-trained checkpoints
- Training recipes
- Modularized components

**Automatic Speech Recognition (ASR)**
- Model architectures and loss functions
- Beamsearch decoders

Pillars where we have preliminary investments, but aren’t able to prioritize for now, due to limited bandwidth:

**Generative audio modeling**
- Text-to-Speech (TTS) and voice/singing conversion
- Audio synthesis w/ differentiable DSP

**Source separation and Speech Enhancement**
- Single/multi-channel data simulation
- Signal processing algorithms
- Pre-trained models
CALL FOR ENGAGEMENT!
Community support has been crucial for the success of TorchAudio!

- Notify us of your use cases, user experiences and other feedback
- Propose new features/projects and engage in discussions (by creating GitHub issues)
- Respond to issues, fix bugs, contribute to major new features.
THANK YOU
02 Source Separation and Speech Enhancement with TorchAudio

Zhaoheng Ni
Research Scientist
Meta Platforms, Inc
TorchAudio aims to provide essential and easy-to-use components of source separation and speech enhancement.
In this tutorial, we will use TorchAudio to...

2.1 Speech separation by ConvTasNet
2.2 Music separation by Hybrid Demucs
2.3 Multi-channel speech enhancement by MVDR beamforming
Scan QR code to access tutorial
OR
Visit https://pytorch.org/audio/main/tutorials/mvdr_tutorial.html
https://pytorch.org/audio/main/tutorials/hybrid_demucs_tutorial.html

Click “Run in Google Colab” or “Download Notebook” up top.
2.1

SPEECH SEPARATION BY CONVTASNET
SPEECH SEPARATION

• Separate mixture of multi-speaker speech into separate tracks.
• The order of speakers can be flexible.
• Energy scale of separated source can be different from target clean speech if using scale-invariant signal-to-distortion (Si-SDR*) loss function.

SPEECH SEPARATION BY CONVTASNET

- TorchAudio provides a pretrained model for ConvTasNet* that is the first neural network outperforms Ideal Ratio Mask in speech separation.
- The model is trained on Libri2Mix dataset with scale-invariant signal-to-distortion (Si-SDR) loss function.


```python
bundle = torchaudio.pipelines.CONVTASNET_BASE_LIBRI2MIX
model = bundle.get_model()
waveform, sample_rate = torchaudio.load("example.wav")
 sources = model(waveform[None])
```
### SPEECH SEPARATION BY CONVTASNET

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Mixture Speech</td>
<td></td>
</tr>
<tr>
<td>Speaker 1 (clean)</td>
<td></td>
</tr>
<tr>
<td>Speaker 1 (estimate)</td>
<td></td>
</tr>
<tr>
<td>Speaker 2 (clean)</td>
<td></td>
</tr>
<tr>
<td>Speaker 2 (estimate)</td>
<td></td>
</tr>
</tbody>
</table>
2.2

MUSIC SEPARATION BY HYBRID DEMUCS
MUSIC SEPARATION

• Separate music waveform into different tracks (vocal, drum, bass, other).
• The order of separated sources are usually fixed.
• Energy scale of separated source is the same as the target clean audio.
MUSIC SEPARATION BY HYBRID DEMUCS

- Hybrid Demucs* uses both time-domain (waveform) and time-frequency domain (spectrogram) as input features.

MUSIC SEPARATION BY HYBRID DEMUCS

TorchAudio provides two pre-trained models for music separation:

- torchaudio.pipelines.HDEMUCS_HIGH_MUSDB
- torchaudio.pipelines.HDEMUCS_HIGH_MUSDB_PLUS

```python
bundle = torchaudio.pipelines.HDEMUCS_HIGH_MUSDB_PLUS
model = bundle.get_model()
```
MUSIC SEPARATION BY HYBRID DEMUCS

- Feed the whole music waveform to HDemucs model may throw OOM error.
- Users can apply music separation chunk by chunk, and concatenate chunks by with faded overlaps.

```python
# Load example mixture music
waveform, sample_rate = torchaudio.load("example.wav")

# Normalize waveform by mean and std
ref = waveform.mean(0)
waveform = (waveform - ref.mean()) / ref.std()

# Apply source separation
sources = model(waveform[None])

# De-normalize separated sources
Sources = sources * ref.std() + ref.mean()
```
MUSIC SEPARATION BY HYBRID DEMUCS

Reference: Drake - Hotline Bling
https://www.youtube.com/watch?v=uxpDa-c-4Mc
2.3

MULTI-CHANNEL SPEECH ENHANCEMENT BY MVDR BEAMFORMING
MULTI-CHANNEL SPEECH ENHANCEMENT BY MVDR BEAMFORMING

- Minimum Variance Distortionless Response (MVDR) Beamforming is one of popular beamforming methods for multi-channel speech enhancement.
MULTI-CHANNEL SPEECH ENHANCEMENT BY MVDR BEAMFORMING

TorchAudio provides essential components for MVDR beamforming:

- `torchaudio.functional.spectrogram` (waveform -> STFT)
- `torchaudio.functional.psd` (compute covariance matrix)
- `torchaudio.functional.mvdr_weights_souden` (compute beamforming weights via Souden* method)
- `torchaudio.functional.apply_beamforming` (estimate enhanced STFT)
- `torchaudio.functional.inverse_spectrogram` (STFT -> waveform)

```
import torchaudio.functional as F

waveform, sample_rate = torchaudio.load("noisy.wav")
length = waveform.shape[-1]

# Compute STFT of mixture speech
stft_mix = F.spectrogram(waveform_mix)

# Compute covariance matrices of speech and noise
psd_speech = F.psd(stft_mix, mask_speech)
psd_noise = F.psd(stft_mix, mask_noise)

# Compute MVDR beamforming weights
w_mvdr = F.mvdr_weights_souden(psd_speech, psd_noise, 0)

# Apply MVDR beamforming
stft_enhanced = F.apply_beamforming(w_mvdr, stft_mix)

# Convert STFT to waveform
waveform_enhanced = F.inverse_spectrogram(stft_enhanced, length)
```

MULTI-CHANNEL SPEECH ENHANCEMENT BY MVDR BEAMFORMING

TorchAudio also provides Module based implementations for MVDR beamforming:

- `torch audio.transforms.Spectrogram` (waveform -> STFT)
- `torch audio.transforms.PSD` (compute covariance matrix)
- `torch audio.transforms.SoudenMVDR` (apply MVDR beamforming and estimate enhanced STFT)
- `torch audio.functional.InverseSpectrogram` (STFT -> waveform)

```python
import torch audio.transforms as T

waveform, sample_rate = torch audio.load("noisy.wav")
length = waveform.shape[-1]

# Initialize modules for beamforming
stft = T.Spectrogram(n_fft=1024, hop_length=256, power=None)
istft = T.InverseSpectrogram(n_freq=1024, hop_length=256)
psd_transform = T.PSD()
mvdr_transform = T.SoudenMVDR()

# Compute STFT of mixture speech
stft_mix = stft(waveform_mix)

# Compute covariance matrices of speech and noise
psd_speech = psd_transform(stft_mix, mask_speech)
psd_noise = psd_transform(stft_mix, mask_noise)

# Apply MVDR beamforming
stft_enhanced = mvdr_transform(stft_mix, psd_speech, psd_noise, 0)

# Convert STFT to waveform
waveform_enhanced = istft(stft_enhanced, length)
```
MULTI-CHANNEL SPEECH ENHANCEMENT BY MVDR BEAMFORMING
THANK YOU
03 Streaming I/O and ASR with TorchAudio

Jeff Hwang
Software Engineer
Meta Platforms, Inc
TorchAudio makes it easy to work with streaming media in PyTorch.
In this tutorial, we will use TorchAudio to...

3.1 Construct a streaming ASR pipeline
3.2 Set up an audio stream
3.3 Perform streaming ASR
Scan QR code to access tutorial
OR
Visit https://pytorch.org/audio/main/tutorials/online_asr_tutorial.html

Click “Run in Google Colab” or “Download Notebook” up top.
3.1

CONSTRUCT A STREAMING ASR PIPELINE
Among numerous other pre-trained models, TorchAudio provides an Emformer* RNN-T Streaming ASR model trained on LibriSpeech. The model is instantiable via a “bundle” that packages a feature extractor, pre-trained model + decoder, and output token processor.

```
bundle = torchaudio.pipelines.EMFORMER_RNNT_BASE_LIBRISPEECH
feature_extractor = bundle.get_streaming_feature_extractor()
decoder = bundle.get_decoder()
token_processor = bundle.get_token_processor()
```

CONSTRUCT A STREAMING ASR PIPELINE

• In streaming ASR, it’s common practice to chunk the input stream into overlapping pieces for sequential processing.
• Each piece comprises a main segment for which the actual prediction will be made, along with a “right context” segment that provides a preview of future data.
• The bundle stores the lengths of the main segment and right context as properties — we’ll use these later.

```python
sample_rate = bundle.sample_rate
segment_length = bundle.segment_length * bundle.hop_length
context_length = bundle.right_context_length * bundle.hop_length

# Sample rate: 16000
# Main segment: 2560 samples (0.16 seconds)
# Right context: 640 samples (0.04 seconds)
```
3.2

SET UP AN AUDIO STREAM
**SET UP AN AUDIO STREAM**

- `torchaudio.io.StreamReader` allows for decoding audio/video streams to PyTorch Tensors, chunk by chunk.
- It supports numerous formats and sources, e.g. local/remote sources, file-like objects, devices.
- Here, we instantiate a `StreamReader` for a LibriVox audio file and add an output stream configured to output `segment_length` frames per chunk with a sample rate of `sample_rate`.

```python
src = "https://download.pytorch.org/torchaudio/tutorial-assets/greatpiratestories_00_various.mp3"
streamer = StreamReader(src)
streamer.add_basic_audio_stream(
    frames_per_chunk=segment_length,
    sample_rate=bundle.sample_rate,
)
streamer.get_src_stream_info(0)

streamer.get_out_stream_info(0)
```

```python
# StreamReaderSourceAudioStream(
#     media_type='audio',
#     codec='mp3',
#     codec_long_name='MP3 (MPEG audio layer 3)',
#     format='fltp',
#     bit_rate=128000,
#     num_frames=0,
#     bits_per_sample=0,
#     metadata={},
#     sample_rate=44100.0,
#     num_channels=2,
# )
```

```python
# StreamReaderOutputStream(
#     source_index=0,
#     filter_description='aresample=16000,aformat=sample_fmts=fltp',
# )
```
SET UP AN AUDIO STREAM

• Emformer RNN-T expects to be fed overlapping chunks of data.
• StreamReader, however, iterates over the source media in a non-overlapping fashion.
• Accordingly, we’ll build a helper class that, for each chunk, caches the last part of the chunk and prepends that part to the next chunk.
3.3

PERFORM STREAMING ASR
PERFORM STREAMING ASR

• Now, we’re ready to run ASR on the stream.
• At a high level, we read chunks one by one from the streamer and run inference on each chunk.
• To ensure that each chunk’s prediction accounts for previous predictions, we maintain and incorporate model state and the last prediction made.
PERFORM STREAMING ASR

We begin by doing the following:
• Instantiate our caching helper class.
• Initialize variables to store model state and our top hypothesis for the last chunk (prediction).
• Retrieve the chunk iterator for our streamer.

cacher = ContextCacher(segment_length, context_length)
state, hypothesis = None, None
stream_iterator = streamer.stream()
PERFORM STREAMING ASR

Now, we iterate over chunks from the stream iterator, and for each chunk, we do the following:

- Generate features using the feature extractor.
- Feed those features along with the previous state and top hypothesis to the model and decoder to produce top-K hypotheses and update the model state.
- Update the top hypothesis and run the token processor on the hypothesis to generate its transcription.
PERFORM STREAMING ASR

What now?

01
Try out internet radio/video streams.

e.g. Wisconsin Public Radio: https://www.wpr.org/live-streaming-direct-streams-internet-radio

02
Stream audio from your computer’s microphone.


03
Use TorchAudio’s pretrained wav2vec 2.0 model and CTC decoder.

RESOURCES

- Docs: pytorch.org/audio
- Github: github.com/pytorch/audio
THANK YOU
04 Self-Supervised Learning (SSL) with TorchAudio

Caroline Chen
Software Engineer
Meta Platforms, Inc
SELF-SUPERVISED LEARNING WITH TORCHAUDIO

TorchAudio aims to provide advanced and flexible SSL infrastructures to facilitate research, R2P, and technology transfer.
AGENDA

4.1
SSL support in TorchAudio

4.2
Basic tutorial using pre-trained utilities

4.3
End-to-end SSL recipe for ASR

4.4
Ongoing and future work
SELF-SUPERVISED LEARNING IN AUDIO

• **Supervised learning** approaches have revolutionized audio and speech processing, but annotating a large amount of data is still a challenge.

• **Self-supervised learning** learns without supervision of labels, making it a promising way to leverage unlabeled data, which the audio domain is full of.
SELF-SUPERVISED LEARNING IN AUDIO

- SSL models are pretrained on a large unlabeled dataset learn audio representations (upstream model)
- Models fine-tune the audio representations on usually limited labeled data (downstream tasks)
- Wav2Vec2.0\(^1\), HuBERT\(^2\) have demonstrated strong success
- Upstream models are usually very expensive to replicate, but can easily be applied to various downstream tasks → pre-trained models facilitate adoption


PYTORCH 2022

SUPERB BENCHMARK

- Framework for evaluating pre-trained SSL models on a variety of downstream speech and audio ML tasks
- Results demonstrate the versatility of SSL models

Abstract
Self-supervised learning (SSL) has proven vital for advancing research in natural language processing (NLP) and computer vision (CV). The paradigm pretrains a shared model on large volumes of unlabeled data and achieves state-of-the-art (SOTA) for various tasks with minimal adaptation. However, the speech processing community lacks a similar setup to systematically explore the paradigm. To bridge this gap, we introduce Speech processing Universal PERformance Benchmark (SUPERB). SUPERB is a leaderboard to benchmark the performance of a shared model across a wide range of speech processing tasks with minimal architecture changes and labeled data. Among multiple usages of the shared model, we especially focus on extracting the representation learned from SSL for its performable re-usability. We present a simple framework to solve SUPERB tasks by training task-specialized lightweight prediction heads on top of the frozen shared model. Our results demonstrate that the framework is promising as SSL representations show competitive generalizability and accessibility across SUPERB tasks. We release SUPERB as a challenge with a leaderboard² and a benchmark toolkit² to fuel the research in representation learning and general speech processing.

Index Terms: Speech, Self-Supervised Learning, Representation Learning, Model Generalization, Benchmark, Evaluation

1. Introduction

Starting from ELMo [1] and BERT [2] in NLP, the effectiveness of SSL is evident in various domains [3, 4]. It is becoming a new principle to solve problems by pretraining a shared model with self-supervision tasks on a large amount of unlabeled data to encode general-purpose knowledge. The model can then be specialized in various downstream tasks through concatenating prediction layers and simple finetuning. This approach achieves SOTA performance in many applications. SSL is desirable for its outstanding performance as well as generalizability and re-usability across tasks to democratize deep learning to various application scenarios. Developing deep neural networks is expensive nowadays in terms of data collection, modeling, computing power, and training time. Repeating the same process for each specific use case is time- and cost-prohibitive for both academic and industrial researchers. SSL can significantly speed up and lower the entry barrier for model development, as the pretrained model is powerful to encode generally applicable knowledge, and only requires low resources to extract task-specific knowledge for different use cases. Well-established benchmark, such as GLUE [5] in NLP and VISSL [6] in CV, is essential to evaluate pretrained models’ generalizability and re-usability across a wide range of tasks.

SSL has been explored in speech, including pretraining with generative loss [7, 8, 9, 10], discriminative loss [11, 12, 13, 14], or multi-task [15, 16]. Researchers have investigated these SSL models’ capabilities on tasks including phoneme classification [11, 17], speaker identification [7, 8], speaker verification [17], emotion recognition [15], ASR [9, 12, 10, 16], speech translation [7], spoken language understanding [18], voice conversion [19] and TTS [20]. While these works showed promising results of SSL on various speech processing tasks, unlike CV or NLP areas, they were investigated with different datasets and experimental setups. Absence of a shared benchmark makes it hard to compare and draw insights across the techniques. Furthermore, existing works explored a limited number of tasks.

4.1

SSL SUPPORT IN TORCHAUDIO
DOWNSHEET DATASET SUPPORT

- Datasets for SUPERB downstream tasks
  - LibriSpeech (ASR, Phoneme Recognition)
  - LibriMix (Speaker Diarization, Source Separation)
  - TF SpeechCommands (Keyword Spotting)
  - FluentSpeechCommands (Intent Classification)
  - VoxCeleb1 (Speaker ID, Speaker Verification)
  - QUESST14 (Query by Example)
  - SNIPS (End-to-end Slot Filling)
  - IEMOCAP (Emotion Recognition)

- Metadata mode support

```python
dataset = torchaudio.datasets.LibriMix(
    root="/home/datasets/",
    subset="test"
)
```

**get_metadata**

Get metadata for the n-th sample from the dataset.

**__getitem__**

Load the n-th sample from the dataset.
MODEL ARCHITECTURE SUPPORT

- Wav2Vec2Model
  - wav2vec2_model()
  - wav2vec2_base()
  - wav2vec2_large()
  - wav2vec2_large_lv60k()
  - hubert_base()
  - hubert_large()
  - hubert_xlarge()

- Support loading fairseq and HuggingFace models into TorchAudio format

```python
# Instantiate a model
model = wav2vec2_base()

# Load model using fairseq
model_file = 'wav2vec_small.pt'
model, _, _ = fairseq.checkpoint_utils.load_model_ensemble_and_task([model_file])
original = model[0]
imported = import_fairseq_model(original)
```
MODEL ARCHITECTURE SUPPORT – CONT

- HuBERTPretrainModel
  - hubert_pretrain_model()
  - hubert_pretrain_base()
  - hubert_pretrain_large()
  - hubert_pretrain_xlarge()
PRETRAINED BUNDLE SUPPORT

- `torchaudio.pipelines`

- **Wav2Vec2Bundle**
  - Pretrained models without fine-tuning

- **Wav2Vec2ASRBundle**
  - Pretrained models fine-tuned for ASR
PRETRAINED BUNDLE SUPPORT

- Wav2Vec2Bundle
  - Wav2Vec2: BASE, LARGE, LARGE_LV60K, XLRS53
  - HuBERT: BASE, LARGE, XLARGE
  - WavLM: BASE, BASE_PLUS, LARGE

- Wav2Vec2ASRBundle
  - Wav2Vec2: BASE, LARGE, LARGE_LV60K arch for 10M, 100H, 960H subsets
  - VoxPopuli: BASE_10K arch for DE, EN, ES, FR, IT subsets
  - HuBERT: LARGE, XLARGE

bundle = torchaudio.pipelines.WAV2VEC2_ASR_BASE_960H
print("Sample Rate:", bundle.sample_rate)
print("Labels:", bundle.get_labels())

Out:
Sample Rate: 16000

model = bundle.get_model().to(device)
print(model.__class__)

Out:
Downloading: "https://download.pytorch.org/torchaudio/models/wav2vec2_fairseq_base_ls960_asr_ls960.pth" to
/root/.cache/torch/hub/checkpoints/wav2vec2_fairseq_base_ls960_asr_ls960.pth

0% | 0.00/360M [00:00<7.78/s]
11% | 0.09/360M [00:03<7.78/s]
23% | 0.17/360M [00:07<7.78/s]
34% | 0.24/360M [00:11<7.78/s]
43% | 0.31/360M [00:15<7.78/s]
53% | 0.38/360M [00:19<7.78/s]
62% | 0.45/360M [00:23<7.78/s]
72% | 0.52/360M [00:27<7.78/s]
81% | 0.59/360M [00:31<7.78/s]
90% | 0.66/360M [00:35<7.78/s]
99% | 0.72/360M [00:39<7.78/s]
100% | 0.78/360M [00:43<7.78/s]

<class 'torchaudio.models.wav2vec2.models.Wav2Vec2Model'>
# Construct decoder

```python
beam_search_decoder = ctc_decoder(
    lexicon="lexicon.txt",
    tokens=['a', 'b', 'c', 'd'],
    lm="kenlm.arpa",
    beam_size=1500,
    lm_weight=2,
)
```

# Compute transcript

```python
results = ctc_decoder(emission)
beam_search_transcript = " " .join(beam_search_result[0][0].words).strip()
```
4.2

BASIC TUTORIAL – ASR USING PRETRAINED SSL MODELS AND UTILITIES
Scan QR code to access tutorial
OR
Visit https://pytorch.org/audio/0.13.0/tutorials/asr_inference_with_ctc_decoder_tutorial.html

Click “Run in Google Colab” or “Download Notebook” up top.
LOAD PRETRAINED BUNDLE AND MODEL

- load in pre-trained Wav2Vec2 bundle that is fine-tuned for ASR, torchaudio.pipelines.WAV2VEC2_ASR_BASE_10M

```python
# Load pretrained bundle and model
bundle = torchaudio.pipelines.WAV2VEC2_ASR_BASE_10M
acoustic_model = bundle.get_model()
```
GET MODEL OUTPUT EMISSIONS

- Load in sample speech file using `torchaudio.load`
- Resample audio waveform to match sample rate of the bundle
- Pass waveform to the acoustic model to get the resulting emission tensor

```python
# Load in and resample waveform
waveform, sample_rate = torchaudio.load(speech_file)

if sample_rate != bundle.sample_rate:
    waveform = torchaudio.functional.resample(waveform, sample_rate, bundle.sample_rate)

# Compute acoustic model output
emission, _ = acoustic_model(waveform)
```
CONSTRUCT CTC DECODER FROM PRETRAINED FILES

- load in token, lexicon, and KenLM language model files for beam search decoding
- construct CTC beam search decoder

```python
from torchaudio.models.decoder import download_pretrained_files
files = download_pretrained_files("librispeech-4-gram")
beam_search_decoder = ctc_decoder(
    lexicon=files.lexicon,
    tokens=files.tokens,
    lm=files.lm
)
```
RUN INFERENCE

- pass emission to the beam search decoder
- parse beam search decoder output
- compute word error rate using torchaudio.functional.edit_distance

```python
beam_search_result = beam_search_decoder(emission)

beam_search_transcript = 
    "".join(beam_search_result[0][0].words).strip()

beam_search_wer = 
torchaudio.functional.edit_distance(actual_transcript,
    beam_search_result[0][0].words) / len(actual_transcript)
```
RESULTS

original transcript:

i really was very much afraid of showing him how much shocked i was at some parts of what he said

greedy decoding:

Out:

Transcript: i reily was very much affrayd of showing him howmuch shoktd i wause at some parte of what he seid
WER: 0.38095238095238093

beam search decoding:

Out:

Transcript: i really was very much afraid of showing him how much shocked i was at some part of what he said
WER: 0.047619047619047616
4.3

END-TO-END SSL RECIPE FOR ASR
Scan QR code to source code
OR
Visit https://github.com/pytorch/audio/tree/main/examples/hubert
HUBERT PRETRAINING RECIPE – PREPROCESSING (FIRST ITERATION)

- Training and validation file list creation
- KMeans clustering model training on MFCC features
- Pseudo-label generation for masked prediction training

```bash
srun \
--cpus-per-task=24 \
python preprocess.py \
--root-dir /home/datasets \
--feat-type mfcc \
--exp-dir ./exp \
--num-cluster 100
```

https://github.com/pytorch/audio/blob/main/examples/hubert/preprocess.py
HUBERT PRETRAINING RECIPE – TRAINING (FIRST ITERATION)

- Trains HuBERTPretrainModel using labels generated by KMeans clustering
- Trained for 250K steps on 32 GPUs
- PyTorch Lightning

```bash
srun \
--gpus-per-node=8 \
--ntasks-per-node=8 \
-N 4 \
--cpus-per-task=10 \
python train.py \
--dataset-path ./exp/data/mfcc/ \
--exp-dir ./exp_iter1 \
--feature-type mfcc \
--num-class 100 \
--max-updates 250000 \
--learning-rate 0.0005 -\n--gpus 8 \
--num-nodes 4
```

https://github.com/pytorch/audio/blob/main/examples/hubert/train.py
HUBERT PRETRAINING RECIPE – PREPROCESSING
(SECOND ITERATION)

- Training and validation file list creation
- KMeans clustering model training on features generated by an intermediate transformer layer of the pretrained HuBERTPretrainModel
- Pseudo-label generation for masked prediction training, for second iteration of pre-training

```bash
srun \
--cpus-per-task=24 \npython preprocess.py \n--root-dir /home/datasets \n--feat-type hubert \n--exp-dir ./exp \n--num-cluster 100
```

https://github.com/pytorch/audio/blob/main/examples/hubert/preprocess.py
HUBERT PRETRAINING RECIPE – TRAINING (SECOND ITERATION)

- Trains HuBERTPretrainModel using labels generated by KMeans clustering
- Trained for 400k steps

```
srun \
--gpus-per-node=8 \
--ntasks-per-node=8 \
-N 4 \
--cpus-per-task=10 \
python train.py \
--dataset-path ./exp/data/mfcc/ \
--exp-dir ./exp_iter1 \
--feature-type mfcc \
--num-class 100 \
--max-updates 400000 \
--learning-rate 0.0005 -\n--gpus 8 \
--num-nodes 4
```

https://github.com/pytorch/audio/blob/main/examples/hubert/train.py
**HUBERT FINETUNING RECIPE**

- Fine-tune HuBERTPretrainModel on LibriLightLimited dataset
- Extra feed-forward layer on top of transformer layers
- Feature extraction layers are frozen during entire process
- Transformer layers are frozen for the first 10K iterations, and then fine-tuned for the rest
- CTC layers are fine-tuned throughout

```python
python finetune.py \
--dataset-path ./root/datasets/ \
--exp-dir ./exp_finetune \
--checkpoint /exp_iter2/checkpoints_librispeech_hubert_pretrain_base/epoch=361-step=399999.ckpt \
--gpus 1 \
--debug \
--warmup-updates 2000 \
--hold-updates 8000 \
--decay-updates 10000 \
--max-updates 20000 \
--learning-rate 5e-5
```

HUBERT EVALUATION

- ASR decoding using CTC beam search decoder

```bash
srun python evaluate.py \
   --librispeech_path /root/datasets/ \
   --checkpoint \
   ./exp_finetune/checkpoints_hubert_pretrain_base/epoch \n   =106-step\=19500.ckpt \
   --split test-clean \
   --use-lm \
   --beam-size 1500 \
   --lm-weight 2.46 \
   --word-score -0.59
```

https://github.com/pytorch/audio/blob/main/examples/hubert/evaluate.py
RESULTS

- fine-tuned HuBERT base on 10h subset of LibriLightLimited dataset

<table>
<thead>
<tr>
<th></th>
<th>WER% (Viterbi)</th>
<th>WER% (KenLM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>dev-clean</td>
<td>10.9</td>
<td>4.2</td>
</tr>
<tr>
<td>dev-other</td>
<td>17.5</td>
<td>9.4</td>
</tr>
<tr>
<td>test-clean</td>
<td>10.9</td>
<td>4.4</td>
</tr>
<tr>
<td>test-other</td>
<td>17.8</td>
<td>9.5</td>
</tr>
</tbody>
</table>
4.4

CURRENT AND FUTURE WORK
ONGOING AND FUTURE WORK

To facilitate research, R2P, and technology transfer around audio SSL.

ADDITIONAL MODEL ARCHITECTURES AND BUNDLES

- Conformer Wav2Vec2
- Emformer HuBERT

MODULARIZED SSL COMPONENTS

- Unified pipeline for SSL
- Modularized components
  - dataset
  - model architecture
  - loss function
THANK YOU!!
RELEVANT LINKS

- Documentation (stable): https://pytorch.org/audio/0.13.0/
- GitHub: https://github.com/pytorch/audio
- Example recipes: https://github.com/pytorch/audio/tree/main/examples

CITING OUR WORK