lhotse

Release 0.12.0.dev

Lhotse development team

May 19, 2022

CONTENTS:

1 Getting started **1**
- 1.1 About . . . 2
- 1.2 Installation . . . 4
- 1.3 Examples . . . 5

2 Representing a corpus **7**
- 2.1 Recording manifest . . . 7
- 2.2 Supervision manifest . . . 9
- 2.3 Standard data preparation recipes . . . 12
- 2.4 Adding new corpora . . . 13

3 Cuts **15**
- 3.1 Overview . . . 15
- 3.2 Types of cuts . . . 20
- 3.3 CLI . . . 21

4 Feature extraction **23**
- 4.1 Storing features . . . 23
- 4.2 Feature normalization . . . 25
- 4.3 Python usage . . . 25
- 4.4 CLI usage . . . 26
- 4.5 Kaldi compatibility caveats . . . 26

5 Executing tasks in parallel **27**

6 PyTorch Datasets **29**
- 6.1 A quick re-cap of PyTorch's data API . . . 29
- 6.2 About Lhotse's Datasets and Samplers . . . 29
- 6.3 Restoring sampler's state: continuing the training . . . 30
- 6.4 Batch I/O: pre-computed vs. on-the-fly features . . . 31
- 6.5 Dataset's list . . . 31
- 6.6 Sampler's list . . . 35
- 6.7 Input strategies' list . . . 35
- 6.8 Augmentation - transforms on cuts . . . 39
- 6.9 Augmentation - transforms on signals . . . 41
- 6.10 Collation utilities for building custom Datasets . . . 44

7 Kaldi Interoperability **49**
- 7.1 Data import/export . . . 49
- 7.2 Kaldi feature extractors . . . 49
- 7.3 Python . . . 49

7.4 CLI 50

8 Command-line interface **51**
8.1 lhotse 51

9 API Reference **103**
9.1 Recording manifests 103
9.2 Supervision manifests 119
9.3 Feature extraction and manifests 130
9.4 Augmentation 255
9.5 Cuts 255
9.6 Recipes 316
9.7 Kaldi conversion 316
9.8 Others 317

10 Indices and tables **319**

Python Module Index **321**

Index **323**

CHAPTER ONE

GETTING STARTED

Lhotse is a Python library aiming to make speech and audio data preparation flexible and accessible to a wider community. Alongside k2, it is a part of the next generation Kaldi speech processing library.

1.1 About

1.1.1 Main goals

- Attract a wider community to speech processing tasks with a **Python-centric design**.
- Accommodate experienced Kaldi users with an **expressive command-line interface**.
- Provide **standard data preparation recipes** for commonly used corpora.
- Provide **PyTorch Dataset classes** for speech and audio related tasks.
- Flexible data preparation for model training with the notion of **audio cuts**.
- **Efficiency**, especially in terms of I/O bandwidth and storage capacity.

1.1.2 Tutorials

We currently have the following tutorials available in *examples* directory: * Basic complete Lhotse workflow * Transforming data with Cuts * *(experimental)* WebDataset integration * How to combine multiple datasets

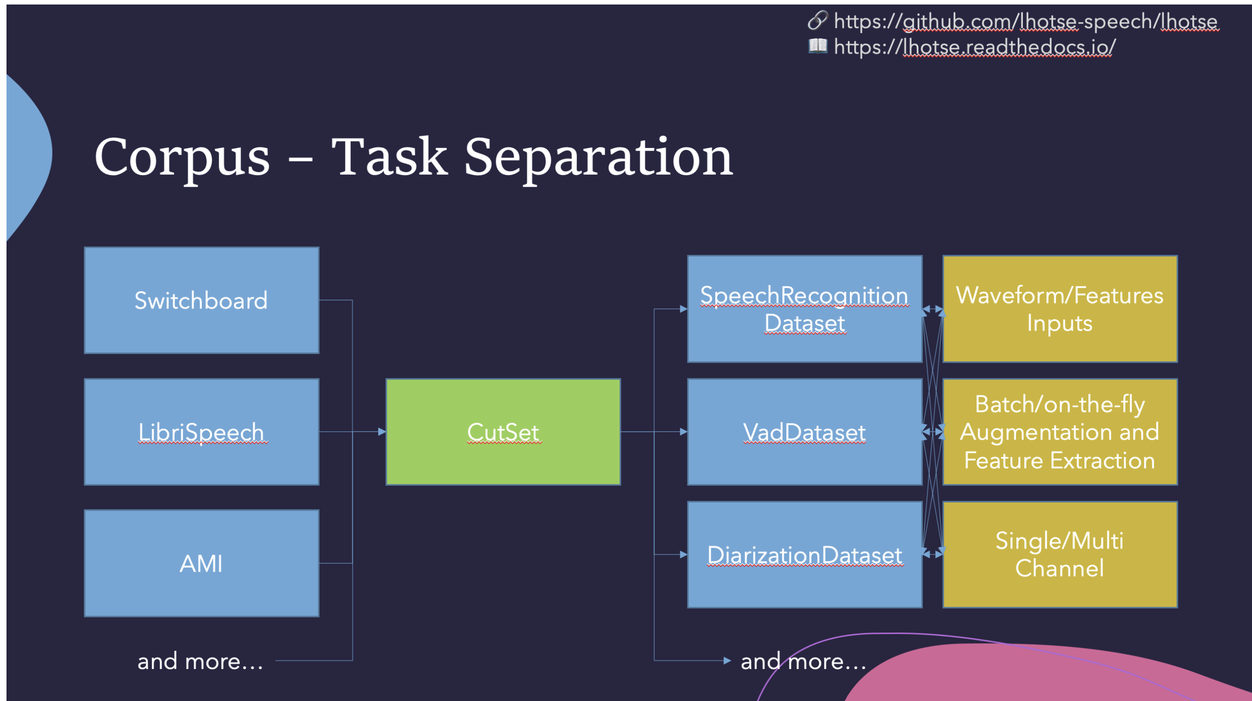
1.1.3 Examples of use

Check out the following links to see how Lhotse is being put to use:

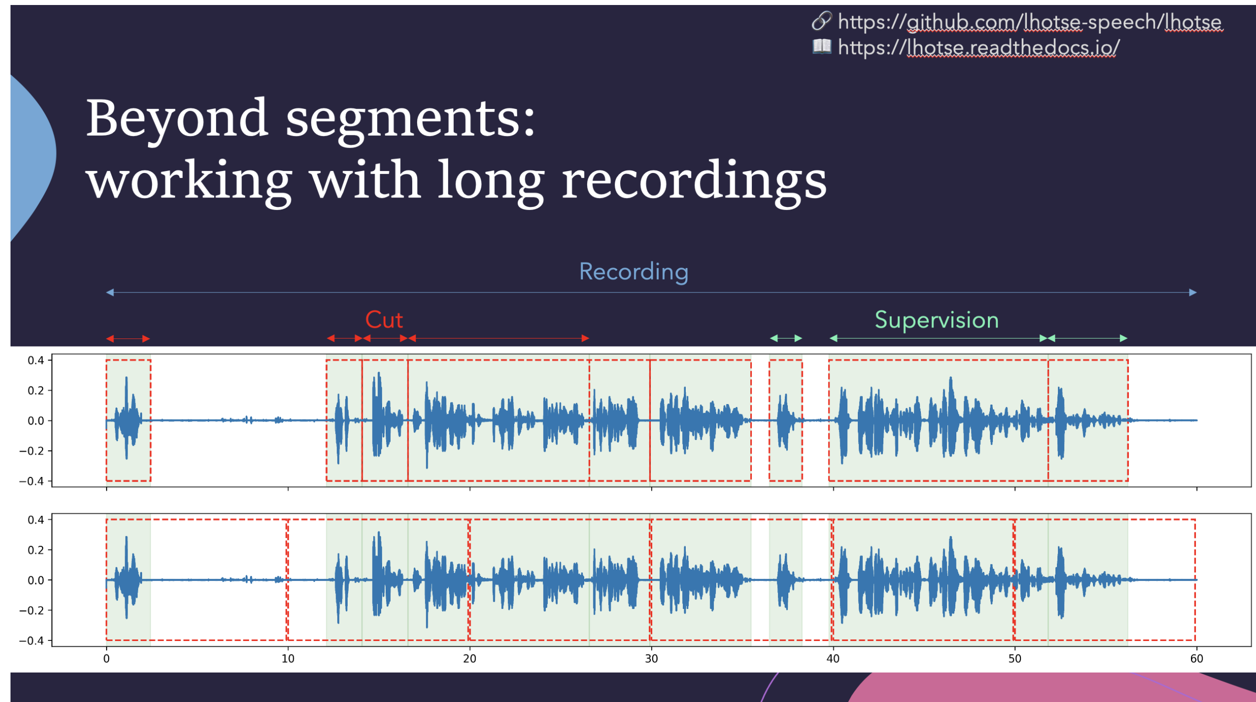
- Icefall recipes: where k2 and Lhotse meet.
- Minimal ESPnet+Lhotse example:

1.1.4 Main ideas

Like Kaldi, Lhotse provides standard data preparation recipes, but extends that with a seamless PyTorch integration through task-specific Dataset classes. The data and meta-data are represented in human-readable text manifests and exposed to the user through convenient Python classes.



Lhotse introduces the notion of audio cuts, designed to ease the training data construction with operations such as mixing, truncation and padding that are performed on-the-fly to minimize the amount of storage required. Data augmentation and feature extraction are supported both in pre-computed mode, with highly-compressed feature matrices stored on disk, and on-the-fly mode that computes the transformations upon request. Additionally, Lhotse introduces feature-space cut mixing to make the best of both worlds.



1.2 Installation

Lhotse supports Python version 3.6 and later.

1.2.1 Pip

Lhotse is available on PyPI:

```
pip install lhotse
```

To install the latest, unreleased version, do:

```
pip install git+https://github.com/lhotse-speech/lhotse
```

Hint: for up to 50% faster reading of JSONL manifests, use: `pip install lhotse[orjson]` *to leverage the* orjson *library.*

1.2.2 Development installation

For development installation, you can fork/clone the GitHub repo and install with pip:

```
git clone https://github.com/lhotse-speech/lhotse
cd lhotse
pip install -e '.[dev]'
pre-commit install  # installs pre-commit hooks with style checks

# Running unit tests
```

(continues on next page)

(continued from previous page)

```
pytest test

# Running linter checks
pre-commit run
```

This is an editable installation (-e option), meaning that your changes to the source code are automatically reflected when importing lhotse (no re-install needed). The [dev] part means you're installing extra dependencies that are used to run tests, build documentation or launch jupyter notebooks.

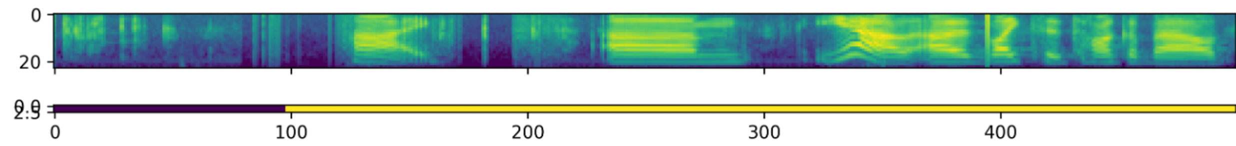
1.3 Examples

We have example recipes showing how to prepare data and load it in Python as a PyTorch `Dataset`. They are located in the `examples` directory.

A short snippet to show how Lhotse can make audio data preparation quick and easy:

```
from torch.utils.data import DataLoader
from lhotse import CutSet, Fbank
from lhotse.dataset import VadDataset, SimpleCutSampler
from lhotse.recipes import prepare_switchboard

# Prepare data manifests from a raw corpus distribution.
# The RecordingSet describes the metadata about audio recordings;
# the sampling rate, number of channels, duration, etc.
# The SupervisionSet describes metadata about supervision segments:
# the transcript, speaker, language, and so on.
swbd = prepare_switchboard('/export/corpora3/LDC/LDC97S62')

# CutSet is the workhorse of Lhotse, allowing for flexible data manipulation.
# We create 5-second cuts by traversing SWBD recordings in windows.
# No audio data is actually loaded into memory or stored to disk at this point.
cuts = CutSet.from_manifests(
    recordings=swbd['recordings'],
    supervisions=swbd['supervisions']
).cut_into_windows(duration=5)

# We compute the log-Mel filter energies and store them on disk;
# Then, we pad the cuts to 5 seconds to ensure all cuts are of equal length,
# as the last window in each recording might have a shorter duration.
# The padding will be performed once the features are loaded into memory.
cuts = cuts.compute_and_store_features(
    extractor=Fbank(),
    storage_path='feats',
    num_jobs=8
).pad(duration=5.0)

# Construct a Pytorch Dataset class for Voice Activity Detection task:
dataset = VadDataset()
sampler = SimpleCutSampler(cuts, max_duration=300)
dataloader = DataLoader(dataset, sampler=sampler, batch_size=None)
batch = next(iter(dataloader))
```

The `VadDataset` will yield a batch with pairs of feature and supervision tensors such as the following - the speech starts roughly at the first second (100 frames):



CHAPTER

TWO

REPRESENTING A CORPUS

In Lhotse, we represent the data using a small number of Python classes, enhanced with methods for solving common data manipulation tasks, that can be stored as JSON or JSONL manifests. For most audio corpora, we will need two types of manifests to fully describe them: a recording manifest and a supervision manifest.

2.1 Recording manifest

class lhotse.audio.**Recording**(*id*, *sources*, *sampling_rate*, *num_samples*, *duration*, *transforms=None*)

The *Recording* manifest describes the recordings in a given corpus. It contains information about the recording, such as its path(s), duration, the number of samples, etc. It allows to represent multiple channels coming from one or more files.

This manifest does not specify any segmentation information or supervision such as the transcript or the speaker – we use *SupervisionSegment* for that.

Note that *Recording* can represent both a single utterance (e.g., in LibriSpeech) and a 1-hour session with multiple channels and speakers (e.g., in AMI). In the latter case, it is partitioned into data suitable for model training using *Cut*.

Hint: Lhotse reads audio recordings using pysoundfile and audioread, similarly to librosa, to support multiple audio formats. For OPUS files we require ffmpeg to be installed.

Hint: Since we support importing Kaldi data dirs, if `wav.scp` contains unix pipes, *Recording* will also handle them correctly.

Examples

A *Recording* can be simply created from a local audio file:

```
>>> from lhotse import RecordingSet, Recording, AudioSource
>>> recording = Recording.from_file('meeting.wav')
>>> recording
Recording(
    id='meeting',
    sources=[AudioSource(type='file', channels=[0], source='meeting.wav')],
    sampling_rate=16000,
    num_samples=57600000,
    duration=3600.0,
```

(continues on next page)

(continued from previous page)

```
    transforms=None
)
```

This manifest can be easily converted to a Python dict and serialized to JSON/JSONL/YAML/etc:

```
>>> recording.to_dict()
{'id': 'meeting',
 'sources': [{'type': 'file',
   'channels': [0],
   'source': 'meeting.wav'}],
 'sampling_rate': 16000,
 'num_samples': 57600000,
 'duration': 3600.0}
```

Recordings can be also created programatically, e.g. when they refer to URLs stored in S3 or somewhere else:

```
>>> s3_audio_files = ['s3://my-bucket/123-5678.flac', ...]
>>> recs = RecordingSet.from_recordings(
...     Recording(
...         id=url.split('/')[-1].replace('.flac', ''),
...         sources=[AudioSource(type='url', source=url, channels=[0])],
...         sampling_rate=16000,
...         num_samples=get_num_samples(url),
...         duration=get_duration(url)
...     )
...     for url in s3_audio_files
... )
```

It allows reading a subset of the audio samples as a numpy array:

```
>>> samples = recording.load_audio()
>>> assert samples.shape == (1, 16000)
>>> samples2 = recording.load_audio(offset=0.5)
>>> assert samples2.shape == (1, 8000)
```

class lhotse.audio.**RecordingSet**(*recordings=None*)

RecordingSet represents a collection of recordings, indexed by recording IDs. It does not contain any annotation such as the transcript or the speaker identity – just the information needed to retrieve a recording such as its path, URL, number of channels, and some recording metadata (duration, number of samples).

It also supports (de)serialization to/from YAML/JSON/etc. and takes care of mapping between rich Python classes and YAML/JSON/etc. primitives during conversion.

When coming from Kaldi, think of it as `wav.scp` on steroids: *RecordingSet* also has the information from *reco2dur* and *reco2num_samples*, is able to represent multi-channel recordings and read a specified subset of channels, and support reading audio files directly, via a unix pipe, or downloading them on-the-fly from a URL (HTTPS/S3/Azure/GCP/etc.).

Examples:

RecordingSet can be created from an iterable of *Recording* objects:

```
>>> from lhotse import RecordingSet
>>> audio_paths = ['123-5678.wav', ...]
>>> recs = RecordingSet.from_recordings(Recording.from_file(p) for p in␣
↪audio_paths)
```

As well as from a directory, which will be scanned recursively for files with parallel processing:

```
>>> recs2 = RecordingSet.from_dir('/data/audio', pattern='*.flac', num_
↪jobs=4)
```

It behaves similarly to a `dict`:

```
>>> '123-5678' in recs
True
>>> recording = recs['123-5678']
>>> for recording in recs:
>>>     pass
>>> len(recs)
127
```

It also provides some utilities for I/O:

```
>>> recs.to_file('recordings.jsonl')
>>> recs.to_file('recordings.json.gz')  # auto-compression
>>> recs2 = RecordingSet.from_file('recordings.jsonl')
```

Manipulation:

```
>>> longer_than_5s = recs.filter(lambda r: r.duration > 5)
>>> first_100 = recs.subset(first=100)
>>> split_into_4 = recs.split(num_splits=4)
>>> shuffled = recs.shuffle()
```

And lazy data augmentation/transformation, that requires to adjust some information in the manifest (e.g., `num_samples` or `duration`). Note that in the following examples, the audio is untouched – the operations are stored in the manifest, and executed upon reading the audio:

```
>>> recs_sp = recs.perturb_speed(factor=1.1)
>>> recs_vp = recs.perturb_volume(factor=2.)
>>> recs_rvb = recs.reverb_rir(rir_recs)
>>> recs_24k = recs.resample(24000)
```

2.2 Supervision manifest

class lhotse.supervision.**SupervisionSegment**(*id*, *recording_id*, *start*, *duration*, *channel=0*, *text=None*, *language=None*, *speaker=None*, *gender=None*, *custom=None*, *alignment=None*)

`SupervisionSegment` represents a time interval (segment) annotated with some supervision labels and/or metadata, such as the transcription, the speaker identity, the language, etc.

Each supervision has unique `id` and always refers to a specific recording (via `recording_id`) and a specific `channel` (by default, 0). It's also characterized by the start time (relative to the beginning of a *Recording* or a *Cut*) and a duration, both expressed in seconds.

The remaining fields are all optional, and their availability depends on specific corpora. Since it is difficult to predict all possible types of metadata, the `custom` field (a dict) can be used to insert types of supervisions that are not supported out of the box.

`SupervisionSegment` may contain multiple types of alignments. The `alignment` field is a dict, indexed by alignment's type (e.g., `word` or `phone`), and contains a list of *`AlignmentItem`* objects – simple structures that contain a given symbol and its time interval. Alignments can be read from CTM files or created programatically.

Examples

A simple segment with no supervision information:

```
>>> from lhotse import SupervisionSegment
>>> sup0 = SupervisionSegment(
...     id='rec00001-sup00000', recording_id='rec00001',
...     start=0.5, duration=5.0, channel=0
... )
```

Typical supervision containing transcript, speaker ID, gender, and language:

```
>>> sup1 = SupervisionSegment(
...     id='rec00001-sup00001', recording_id='rec00001',
...     start=5.5, duration=3.0, channel=0,
...     text='transcript of the second segment',
...     speaker='Norman Dyhrentfurth', language='English', gender='M'
... )
```

Two supervisions denoting overlapping speech on two separate channels in a microphone array/multiple headsets (pay attention to `start`, `duration`, and `channel`):

```
>>> sup2 = SupervisionSegment(
...     id='rec00001-sup00002', recording_id='rec00001',
...     start=15.0, duration=5.0, channel=0,
...     text="i have incredibly good news for you",
...     speaker='Norman Dyhrentfurth', language='English', gender='M'
... )
>>> sup3 = SupervisionSegment(
...     id='rec00001-sup00003', recording_id='rec00001',
...     start=18.0, duration=3.0, channel=1,
...     text="say what",
...     speaker='Hervey Arman', language='English', gender='M'
... )
```

A supervision with a phone alignment:

```
>>> from lhotse.supervision import AlignmentItem
>>> sup4 = SupervisionSegment(
...     id='rec00001-sup00004', recording_id='rec00001',
...     start=33.0, duration=1.0, channel=0,
...     text="ice",
...     speaker='Maryla Zechariah', language='English', gender='F'
...     alignment={
...         'phone': [
...             AlignmentItem(symbol='AY0', start=33.0, duration=0.6),
...             AlignmentItem(symbol='S', start=33.6, duration=0.4)
```

(continues on next page)

(continued from previous page)

```
...         ]
...     }
... )
```

Converting `SupervisionSegment` to a `dict`:

```
>>> sup0.to_dict()
{'id': 'rec00001-sup00000', 'recording_id': 'rec00001', 'start': 0.5,
↪'duration': 5.0, 'channel': 0}
```

class lhotse.supervision.**SupervisionSet**(*segments*)

SupervisionSet represents a collection of segments containing some supervision information (see *SupervisionSegment*), that are indexed by segment IDs.

It acts as a Python `dict`, extended with an efficient `find` operation that indexes and caches the supervision segments in an interval tree. It allows to quickly find supervision segments that correspond to a specific time interval.

When coming from Kaldi, think of *SupervisionSet* as a `segments` file on steroids, that may also contain *text*, *utt2spk*, *utt2gender*, *utt2dur*, etc.

Examples

Building a *SupervisionSet*:

```
>>> from lhotse import SupervisionSet, SupervisionSegment
>>> sups = SupervisionSet.from_segments([SupervisionSegment(...), ...])
```

Writing/reading a *SupervisionSet*:

```
>>> sups.to_file('supervisions.jsonl.gz')
>>> sups2 = SupervisionSet.from_file('supervisions.jsonl.gz')
```

Using *SupervisionSet* like a dict:

```
>>> 'rec00001-sup00000' in sups
True
>>> sups['rec00001-sup00000']
SupervisionSegment(id='rec00001-sup00000', recording_id='rec00001', start=0.
↪5, ...)
>>> for segment in sups:
...     pass
```

Searching by `recording_id` and time interval:

```
>>> matched_segments = sups.find(recording_id='rec00001', start_after=17.0,␣
↪end_before=25.0)
```

Manipulation:

```
>>> longer_than_5s = sups.filter(lambda s: s.duration > 5)
>>> first_100 = sups.subset(first=100)
>>> split_into_4 = sups.split(num_splits=4)
>>> shuffled = sups.shuffle()
```

2.3 Standard data preparation recipes

We provide a number of standard data preparation recipes. By that, we mean a collection of a Python function + a CLI tool that create the manifests given a corpus directory.

Table 1: Currently supported corpora

Corpus name	Function
ADEPT	`lhotse.recipes.prepare_adept()`
Aidatatang_200zh	`lhotse.recipes.prepare_aidatatang_200zh()`
Aishell	`lhotse.recipes.prepare_aishell()`
AISHELL-4	`lhotse.recipes.prepare_aishell4()`
AliMeeting	`lhotse.recipes.prepare_alimeeting()`
AMI	`lhotse.recipes.prepare_ami()`
ASpIRE	`lhotse.recipes.prepare_aspire()`
BABEL	`lhotse.recipes.prepare_single_babel_language()`
BVCC / VoiceMOS Challenge	`lhotse.recipes.bvcc()`
CallHome Egyptian	`lhotse.recipes.prepare_callhome_egyptian()`
CallHome English	`lhotse.recipes.prepare_callhome_english()`
CMU Arctic	`lhotse.recipes.prepare_cmu_arctic()`
CMU Indic	`lhotse.recipes.prepare_cmu_indic()`
CMU Kids	`lhotse.recipes.prepare_cmu_kids()`
CommonVoice	`lhotse.recipes.prepare_commonvoice()`
CSLU Kids	`lhotse.recipes.prepare_cslu_kids()`
DIHARD III	`lhotse.recipes.prepare_dihard3()`
English Broadcast News 1997	`lhotse.recipes.prepare_broadcast_news()`
Fisher English Part 1, 2	`lhotse.recipes.prepare_fisher_english()`
Fisher Spanish	`lhotse.recipes.prepare_fisher_spanish()`
GALE Arabic Broadcast Speech	`lhotse.recipes.prepare_gale_arabic()`
GALE Mandarin Broadcast Speech	`lhotse.recipes.prepare_gale_mandarin()`
GigaSpeech	`lhotse.recipes.prepare_gigaspeech()`
Heroico	`lhotse.recipes.prepare_heroico()`
HiFiTTS	`lhotse.recipes.prepare_hifitts()`
ICSI	`lhotse.recipes.prepare_icsi()`
L2 Arctic	`lhotse.recipes.prepare_l2_arctic()`
LibriCSS	`lhotse.recipes.prepare_libricss()`
LibriSpeech (including "mini")	`lhotse.recipes.prepare_librispeech()`
LibriTTS	`lhotse.recipes.prepare_libritts()`
LJ Speech	`lhotse.recipes.prepare_ljspeech()`
MiniLibriMix	`lhotse.recipes.prepare_librimix()`
MTEDx	`lhotse.recipes.prepare_mtdex()`
MobvoiHotWord	`lhotse.recipes.prepare_mobvoihotwords()`
Multilingual LibriSpeech (MLS)	`lhotse.recipes.prepare_mls()`
MUSAN	`lhotse.recipes.prepare_musan()`
National Speech Corpus (Singaporean English)	`lhotse.recipes.prepare_nsc()`
People's Speech	`lhotse.recipes.prepare_peoples_speech()`
RIRs and Noises Corpus (OpenSLR 28)	`lhotse.recipes.prepare_rir_noise()`
SPGISpeech	`lhotse.recipes.prepare_spgispeech()`
Switchboard	`lhotse.recipes.prepare_switchboard()`
TED-LIUM v3	`lhotse.recipes.prepare_tedlium()`

continues on next page

Table 1 – continued from previous page

Corpus name	Function
TIMIT	`lhotse.recipes.prepare_timit()`
VCTK	`lhotse.recipes.prepare_vctk()`
VoxCeleb	`lhotse.recipes.prepare_voxceleb()`
WenetSpeech	`lhotse.recipes.prepare_wenet_speech()`
YesNo	`lhotse.recipes.prepare_yesno()`
Eval2000	`lhotse.recipes.prepare_eval2000()`
MGB2	`lhotse.recipes.prepare_mgb2()`

2.4 Adding new corpora

Hint: Python data preparation recipes. Each corpus has a dedicated Python file in `lhotse/recipes`, which you can use as the basis for your own recipe.

Hint: (optional) Downloading utility. For publicly available corpora that can be freely downloaded, we usually define a function called `download_<corpus-name>()`.

Hint: Data preparation Python entry-point. Each data preparation recipe should expose a single function called `prepare_<corpus-name>`, that produces dicts like: `{'recordings': <RecordingSet>, 'supervisions': <SupervisionSet>}`.

Hint: CLI recipe wrappers. We provide a command-line interface that wraps the `download` and `prepare` functions – see `lhotse/bin/modes/recipes` for examples of how to do it.

Hint: Pre-defined train/dev/test splits. When a corpus defines standard split (e.g. train/dev/test), we return a dict with the following structure: `{'train': {'recordings': <RecordingSet>, 'supervisions': <SupervisionSet>}, 'dev': ...}`

Hint: Manifest naming convention. The default naming convention is `<corpus-name>_<manifest-type>_<split>.jsonl.gz`, i.e., we save the manifests in a compressed JSONL file. Here, `<manifest-type>` can be `recordings`, `supervisions`, etc., and `<split>` can be `train`, `dev`, `test`, etc. In case the corpus has no such split defined, we can use `all` as default. Other information, e.g., mic type, language, etc. may be included in the `<corpus-name>`. Some examples are: `cmu-indic_recordings_all.jsonl.gz`, `ami-ihm_supervisions_dev.jsonl.gz`, `mtedx-english_recordings_train.jsonl.gz`.

Hint: Isolated utterance corpora. Some corpora (like LibriSpeech) come with pre-segmented recordings. In these cases, the *`SupervisionSegment`* will exactly match the `Recording` duration (and there will likely be exactly one segment corresponding to any recording).

Hint: Conversational corpora. Corpora with longer recordings (e.g. conversational, like Switchboard) should have exactly one *Recording* object corresponding to a single conversation/session, that spans its whole duration. Each speech segment in that recording should be represented as a *SupervisionSegment* with the same `recording_id` value.

Hint: Multi-channel corpora. Corpora with multiple channels for each session (e.g. AMI) should have a single *Recording* with multiple *AudioSource* objects – each corresponding to a separate channel. Remember to make the *SupervisionSegment* objects correspond to the right channels!

CHAPTER

THREE

CUTS

3.1 Overview

Audio cuts are one of the main Lhotse features. Cut is a part of a recording, but it can be longer than a supervision segment, or even span multiple segments. The regions without a supervision are just audio that we don't assume we know anything about - there may be silence, noise, non-transcribed speech, etc. Task-specific datasets can leverage this information to generate masks for such regions.

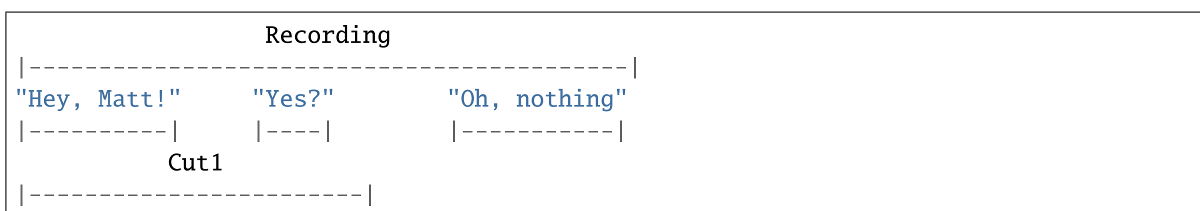
class lhotse.cut.**Cut**

> **Caution:** *Cut* is just an abstract class – the actual logic is implemented by its child classes (scroll down for references).

Cut is a base class for audio cuts. An "audio cut" is a subset of a *Recording* – it can also be thought of as a "view" or a pointer to a chunk of audio. It is not limited to audio data – cuts may also point to (sub-spans of) precomputed *Features*.

Cuts are different from *SupervisionSegment* in that they may be arbitrarily longer or shorter than supervisions; cuts may even contain multiple supervisions for creating contextual training data, and unsupervised regions that provide real or synthetic acoustic background context for the supervised segments.

The following example visualizes how a cut may represent a part of a single-channel recording with two utterances and some background noise in between:



```
                  Recording
|-------------------------------------------|
"Hey, Matt!"     "Yes?"        "Oh, nothing"
|----------|     |----|        |-----------|
           Cut1
|------------------------|
```

This scenario can be represented in code, using *MonoCut*, as:

```
>>> from lhotse import Recording, SupervisionSegment, MonoCut
>>> rec = Recording(id='rec1', duration=10.0, sampling_rate=8000, num_samples=80000,
↪ sources=[...])
>>> sups = [
...     SupervisionSegment(id='sup1', recording_id='rec1', start=0, duration=3.37,␣
↪text='Hey, Matt!'),
...     SupervisionSegment(id='sup2', recording_id='rec1', start=4.5, duration=0.9,␣
↪text='Yes?'),
```

(continues on next page)

(continued from previous page)

```
...     SupervisionSegment(id='sup3', recording_id='rec1', start=6.9, duration=2.9,␣
↪text='Oh, nothing'),
... ]
>>> cut = MonoCut(id='rec1-cut1', start=0.0, duration=6.0, channel=0, recording=rec,
...     supervisions=[sups[0], sups[1]])
```

Note: All Cut classes assume that the *SupervisionSegment* time boundaries are relative to the beginning of the cut. E.g. if the underlying *Recording* starts at 0s (always true), the cut starts at 100s, and the SupervisionSegment inside the cut starts at 3s, it really did start at 103rd second of the recording. In some cases, the supervision might have a negative start, or a duration exceeding the duration of the cut; this means that the supervision in the recording extends beyond the cut.

Cut allows to check and read audio data or features data:

```
>>> assert cut.has_recording
>>> samples = cut.load_audio()
>>> if cut.has_features:
...     feats = cut.load_features()
```

It can be visualized, and listened to, inside Jupyter Notebooks:

```
>>> cut.plot_audio()
>>> cut.play_audio()
>>> cut.plot_features()
```

Cuts can be used with Lhotse's *FeatureExtractor* to compute features.

```
>>> from lhotse import Fbank
>>> feats = cut.compute_features(extractor=Fbank())
```

It is also possible to use a *FeaturesWriter* to store the features and attach their manifest to a copy of the cut:

```
>>> from lhotse import LilcomChunkyWriter
>>> with LilcomChunkyWriter('feats.lca') as storage:
...     cut_with_feats = cut.compute_and_store_features(
...         extractor=Fbank(),
...         storage=storage
...     )
```

Cuts have several methods that allow their manipulation, transformation, and mixing. Some examples (see the respective methods documentation for details):

```
>>> cut_2_to_4s = cut.truncate(offset=2, duration=2)
>>> cut_padded = cut.pad(duration=10.0)
>>> cut_extended = cut.extend_by(duration=5.0, direction='both')
>>> cut_mixed = cut.mix(other_cut, offset_other_by=5.0, snr=20)
>>> cut_append = cut.append(other_cut)
>>> cut_24k = cut.resample(24000)
>>> cut_sp = cut.perturb_speed(1.1)
>>> cut_vp = cut.perturb_volume(2.)
>>> cut_rvb = cut.reverb_rir(rir_recording)
```

Note: All cut transformations are performed lazily, on-the-fly, upon calling `load_audio` or `load_features`. The stored waveforms and features are untouched.

Caution: Operations on cuts are not mutating – they return modified copies of `Cut` objects, leaving the original object unmodified.

A `Cut` that contains multiple segments (`SupervisionSegment`) can be decayed into smaller cuts that correspond directly to supervisions:

```
>>> smaller_cuts = cut.trim_to_supervisions()
```

Cuts can be detached from parts of their metadata:

```
>>> cut_no_feat = cut.drop_features()
>>> cut_no_rec = cut.drop_recording()
>>> cut_no_sup = cut.drop_supervisions()
```

Finally, cuts provide convenience methods to compute feature frame and audio sample masks for supervised regions:

```
>>> sup_frames = cut.supervisions_feature_mask()
>>> sup_samples = cut.supervisions_audio_mask()
```

See also:

- *lhotse.cut.MonoCut*
- *lhotse.cut.MixedCut*
- *lhotse.cut.CutSet*

class lhotse.cut.**CutSet**(*cuts=None*)

`CutSet` represents a collection of cuts, indexed by cut IDs. CutSet ties together all types of data – audio, features and supervisions, and is suitable to represent training/dev/test sets.

Note: `CutSet` is the basic building block of PyTorch-style Datasets for speech/audio processing tasks.

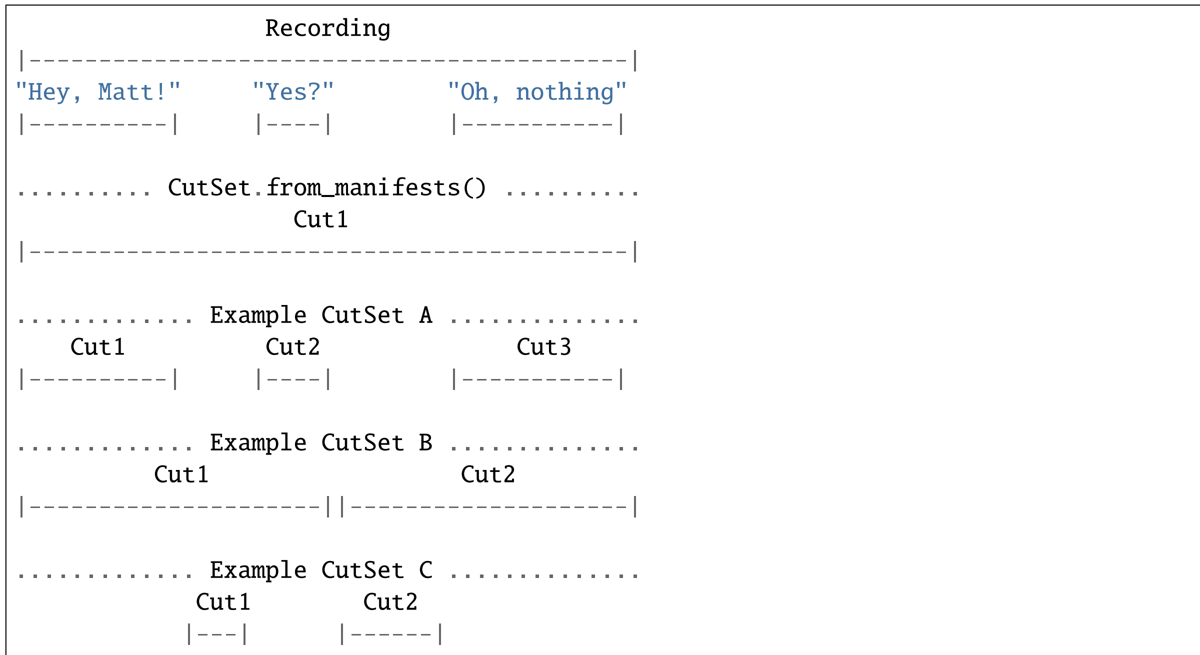
When coming from Kaldi, there is really no good equivalent – the closest concept may be Kaldi's "egs" for training neural networks, which are chunks of feature matrices and corresponding alignments used respectively as inputs and supervisions. `CutSet` is different because it provides you with all kinds of metadata, and you can select just the interesting bits to feed them to your models.

`CutSet` can be created from any combination of `RecordingSet`, `SupervisionSet`, and `FeatureSet` with *lhotse.cut.CutSet.from_manifests()*:

```
>>> from lhotse import CutSet
>>> cuts = CutSet.from_manifests(recordings=my_recording_set)
>>> cuts2 = CutSet.from_manifests(features=my_feature_set)
>>> cuts3 = CutSet.from_manifests(
...     recordings=my_recording_set,
...     features=my_feature_set,
...     supervisions=my_supervision_set,
... )
```

When creating a *CutSet* with *CutSet.from_manifests()*, the resulting cuts will have the same duration as the input recordings or features. For long recordings, it is not viable for training. We provide several methods to transform the cuts into shorter ones.

Consider the following scenario:



```
                  Recording
|-------------------------------------------|
"Hey, Matt!"     "Yes?"        "Oh, nothing"
|----------|     |----|        |-----------|

.......... CutSet.from_manifests() .........
                    Cut1
|-------------------------------------------|

............. Example CutSet A ..............
    Cut1          Cut2              Cut3
|----------|     |----|        |-----------|

............. Example CutSet B ..............
          Cut1                  Cut2
|---------------------||--------------------|

............. Example CutSet C ..............
             Cut1        Cut2
            |---|       |------|
```

The CutSet's A, B and C can be created like:

```
>>> cuts_A = cuts.trim_to_supervisions()
>>> cuts_B = cuts.cut_into_windows(duration=5.0)
>>> cuts_C = cuts.trim_to_unsupervised_segments()
```

Note: Some operations support parallel execution via an optional `num_jobs` parameter. By default, all processing is single-threaded.

Caution: Operations on cut sets are not mutating – they return modified copies of *CutSet* objects, leaving the original object unmodified (and all of its cuts are also unmodified).

CutSet can be stored and read from JSON, JSONL, etc. and supports optional gzip compression:

```
>>> cuts.to_file('cuts.jsonl.gz')
>>> cuts4 = CutSet.from_file('cuts.jsonl.gz')
```

It behaves similarly to a `dict`:

```
>>> 'rec1-1-0' in cuts
True
>>> cut = cuts['rec1-1-0']
>>> for cut in cuts:
>>>     pass
```

(continues on next page)

(continued from previous page)

```
>>> len(cuts)
127
```

CutSet has some convenience properties and methods to gather information about the dataset:

```
>>> ids = list(cuts.ids)
>>> speaker_id_set = cuts.speakers
>>> # The following prints a message:
>>> cuts.describe()
Cuts count: 547
Total duration (hours): 326.4
Speech duration (hours): 79.6 (24.4%)
***
Duration statistics (seconds):
mean    2148.0
std      870.9
min      477.0
25%     1523.0
50%     2157.0
75%     2423.0
max     5415.0
dtype: float64
```

Manipulation examples:

```
>>> longer_than_5s = cuts.filter(lambda c: c.duration > 5)
>>> first_100 = cuts.subset(first=100)
>>> split_into_4 = cuts.split(num_splits=4)
>>> shuffled = cuts.shuffle()
>>> random_sample = cuts.sample(n_cuts=10)
>>> new_ids = cuts.modify_ids(lambda c: c.id + '-newid')
```

These operations can be composed to implement more complex operations, e.g. bucketing by duration:

```
>>> buckets = cuts.sort_by_duration().split(num_splits=30)
```

Cuts in a CutSet can be detached from parts of their metadata:

```
>>> cuts_no_feat = cuts.drop_features()
>>> cuts_no_rec = cuts.drop_recordings()
>>> cuts_no_sup = cuts.drop_supervisions()
```

Sometimes specific sorting patterns are useful when a small CutSet represents a mini-batch:

```
>>> cuts = cuts.sort_by_duration(ascending=False)
>>> cuts = cuts.sort_like(other_cuts)
```

CutSet offers some batch processing operations:

```
>>> cuts = cuts.pad(num_frames=300)  # or duration=30.0
>>> cuts = cuts.truncate(max_duration=30.0, offset_type='start')  # truncate from␣
↪start to 30.0s
>>> cuts = cuts.mix(other_cuts, snr=[10, 30], mix_prob=0.5)
```

CutSet supports lazy data augmentation/transformation methods which require adjusting some information in the manifest (e.g., num_samples or duration). Note that in the following examples, the audio is untouched – the operations are stored in the manifest, and executed upon reading the audio:

```
>>> cuts_sp = cuts.perturb_speed(factor=1.1)
>>> cuts_vp = cuts.perturb_volume(factor=2.)
>>> cuts_24k = cuts.resample(24000)
>>> cuts_rvb = cuts.reverb_rir(rir_recordings)
```

Caution: If the *CutSet* contained *Features* manifests, they will be detached after performing audio augmentations such as *CutSet.perturb_speed()*, *CutSet.resample()*, *CutSet.perturb_volume()*, or *CutSet.reverb_rir()*.

CutSet offers parallel feature extraction capabilities (see *meth*:.CutSet.compute_and_store_features: for details), and can be used to estimate global mean and variance:

```
>>> from lhotse import Fbank
>>> cuts = CutSet()
>>> cuts = cuts.compute_and_store_features(
...     extractor=Fbank(),
...     storage_path='/data/feats',
...     num_jobs=4
... )
>>> mvn_stats = cuts.compute_global_feature_stats('/data/features/mvn_stats.pkl',␣
↪max_cuts=10000)
```

See also:

- *Cut*

3.2 Types of cuts

There are three cut classes: *MonoCut*, *MixedCut*, and *PaddingCut* that are described below in more detail:

class lhotse.cut.**MonoCut**(*id*, *start*, *duration*, *channel*, *supervisions=<factory>*, *features=None*, *recording=None*, *custom=None*)

MonoCut is a *Cut* of a single channel of a *Recording*. In addition to Cut, it has a specified channel attribute. This is the most commonly used type of cut.

Please refer to the documentation of *Cut* to learn more about using cuts.

See also:

- *lhotse.cut.Cut*
- *lhotse.cut.MixedCut*
- *lhotse.cut.CutSet*

class lhotse.cut.**MixedCut**(*id*, *tracks*)

MixedCut is a *Cut* that actually consists of multiple other cuts. It can be interpreted as a multi-channel cut, but its primary purpose is to allow time-domain and feature-domain augmentation via mixing the training cuts with noise, music, and babble cuts. The actual mixing operations are performed on-the-fly.

Internally, *MixedCut* holds other cuts in multiple trakcs (*MixTrack*), each with its own offset and SNR that is relative to the first track.

Please refer to the documentation of *Cut* to learn more about using cuts.

In addition to methods available in *Cut*, *MixedCut* provides the methods to read all of its tracks audio and features as separate channels:

```
>>> cut = MixedCut(...)
>>> mono_features = cut.load_features()
>>> assert len(mono_features.shape) == 2
>>> multi_features = cut.load_features(mixed=False)
>>> # Now, the first dimension is the channel.
>>> assert len(multi_features.shape) == 3
```

See also:

- *lhotse.cut.Cut*
- *lhotse.cut.MonoCut*
- *lhotse.cut.CutSet*

class lhotse.cut.**PaddingCut**(*id, duration, sampling_rate, feat_value, num_frames=None, num_features=None, frame_shift=None, num_samples=None, custom=None*)

PaddingCut is a dummy *Cut* that doesn't refer to actual recordings or features –it simply returns zero samples in the time domain and a specified features value in the feature domain. Its main role is to be appended to other cuts to make them evenly sized.

Please refer to the documentation of *Cut* to learn more about using cuts.

See also:

- *lhotse.cut.Cut*
- *lhotse.cut.MonoCut*
- *lhotse.cut.MixedCut*
- *lhotse.cut.CutSet*

3.3 CLI

We provide a limited CLI to manipulate Lhotse manifests. Some examples of how to perform manipulations in the terminal:

```
# Reject short segments
lhotse yaml filter 'duration>=3.0' cuts.jsonl cuts-3s.jsonl
# Pad short segments to 5 seconds.
lhotse cut pad --duration 5.0 cuts-3s.jsonl cuts-5s-pad.jsonl
# Truncate longer segments to 5 seconds.
lhotse cut truncate --max-duration 5.0 --offset-type random cuts-5s-pad.jsonl cuts-5s.
↪jsonl
```

CHAPTER

FOUR

FEATURE EXTRACTION

Lhotse provides the following feature extractor implementations:

- Log-Mel filter-bank *Fbank* and MFCC *Mfcc* PyTorch implementations. They are very close to Kaldi's, and their underlying components are PyTorch modules that can be used as layers in neural networks (i.e. support batching, GPUs, autograd, and TorchScript). These classes are found in `lhotse.features.kaldi.layers` (in particular: *Wav2LogFilterBank* and *Wav2MFCC*). We also provide online inference methods to support deployment in audio streaming applications.
- Torchaudio Kaldi-compatible extractors *TorchaudioFbank*, *TorchaudioMfcc*, and *Spectrogram*. They only support processing one utterance at a time (batching is not possible).
- Librosa compatible filter-bank feature extractor *LibrosaFbank* (compatible with the one used in ESPnet and ParallelWaveGAN projects for TTS and vocoders);
- kaldifeat – another Kaldi-compatible feature extraction implementation that can process batches of uneven lengths efficiently, implemented in C++ with Python wrappers.
- **`opensmile`_** – a wrapper over popular set of feature extractors, often used in modeling non-verbal aspects of speech (e.g., emotion recognition).

We also support custom defined feature extractors via a Python API.

We are striving for a simple relation between the audio duration, the number of frames, and the frame shift (with a known sampling rate):

```
num_samples = round(duration * sampling_rate)
window_hop = round(frame_shift * sampling_rate)
num_frames = int((num_samples + window_hop // 2) // window_hop)
```

This is equivalent of having Kaldi's `snip_edges` parameter set to False, and Lhotse expects **every** feature extractor to conform to that requirement.

4.1 Storing features

Features in Lhotse are stored as numpy matrices with shape `(num_frames, num_features)`. By default, we use lilcom for lossy compression and reduce the size on the disk by about 3x. The lilcom compression method uses a fixed precision that doesn't depend on the magnitude of the thing being compressed, so it's better suited to log-energy features than energy features. By default, we store these matrices in archives with our own custom format that allows efficient reads of chunks compressed with lilcom. Other options such as HDF5 are also available.

There are two types of manifests:

- one describing the feature extractor;
- one describing the extracted feature matrices.

The feature extractor manifest is mapped to a Python configuration dataclass. An example for *spectrogram*:


```yaml
dither: 0.0
energy_floor: 1e-10
frame_length: 0.025
frame_shift: 0.01
min_duration: 0.0
preemphasis_coefficient: 0.97
raw_energy: true
remove_dc_offset: true
round_to_power_of_two: true
window_type: povey
type: spectrogram
```


And the corresponding configuration class:

class lhotse.features.**SpectrogramConfig**(*dither=0.0*, *window_type='povey'*, *frame_length=0.025*, *frame_shift=0.01*, *remove_dc_offset=True*, *round_to_power_of_two=True*, *energy_floor=1e-10*, *min_duration=0.0*, *preemphasis_coefficient=0.97*, *raw_energy=True*)

dither: float = 0.0

window_type: str = 'povey'

frame_length: float = 0.025

frame_shift: float = 0.01

remove_dc_offset: bool = True

round_to_power_of_two: bool = True

energy_floor: float = 1e-10

min_duration: float = 0.0

preemphasis_coefficient: float = 0.97

raw_energy: bool = True

to_dict()

Return type Dict[str, Any]

static from_dict(*data*)

Return type *SpectrogramConfig*

__init__(*dither=0.0*, *window_type='povey'*, *frame_length=0.025*, *frame_shift=0.01*, *remove_dc_offset=True*, *round_to_power_of_two=True*, *energy_floor=1e-10*, *min_duration=0.0*, *preemphasis_coefficient=0.97*, *raw_energy=True*)

The feature matrices manifest is a list of documents. These documents contain the information necessary to tie the features to a particular recording: `start`, `duration`, `channel` and `recording_id`. They also provide some useful information, such as the type of features, number of frames and feature dimension. Finally, they specify how the feature matrix is stored with `storage_type` (currently `numpy` or `lilcom`), and where to find it with the `storage_path`. In the future there might be more storage types.

```
- channels: 0
  duration: 16.04
  num_features: 23
  num_frames: 1604
  recording_id: recording-1
  start: 0.0
  storage_path: test/fixtures/libri/storage/dc2e0952-f2f8-423c-9b8c-f5481652ee1d.llc
  storage_type: lilcom
  type: fbank
```


4.2 Feature normalization

We will briefly discuss how to perform mean and variance normalization (a.k.a. CMVN) in Lhotse effectively. We compute and store unnormalized features, and it is up to the user to normalize them if they want to do so. There are three common ways to perform feature normalization:

- **Global normalization**: we compute the means and variances using the whole data (`FeatureSet` or `CutSet`), and apply the same transform on every sample. The global statistics can be computed efficiently with `FeatureSet.compute_global_stats()` or `CutSet.compute_global_feature_stats()`. They use an iterative algorithm that does not require loading the whole dataset into memory.
- **Per-instance normalization**: we compute the means and variances separately for each data sample (i.e. a single feature matrix). Each feature matrix undergoes a different transform. This approach seems to be common in computer vision modeling.
- **Sliding window ("online") normalization**: we compute the means and variances using a slice of the feature matrix with a specified duration, e.g. 3 seconds (a standard value in Kaldi). This is useful when we expect the model to work on incomplete inputs, e.g. streaming speech recognition. We currently recommend using Torchaudio CMVN for that.

4.3 Python usage

Typically you'll want to extract features from cuts. In case of long recordings, it is fine to extract the features for long-recording cuts, and cut those into shorter segments later. Our default feature storage mechanism is fairly efficient when reading chunks.

```python
from lhotse import CutSet

cuts = CutSet.from_file("data/cuts.jsonl.gz")
# Create a log Mel energy filter bank feature extractor with default settings
fbank = Fbank()
# Compute features for cuts with 8 parallel jobs and return a new CutSet which
# references those features.
cuts = cuts.compute_and_store_features(
    extractor=fbank,
    storage_path="data/fbank",
    num_jobs=8,
)
cuts.to_file("data/cuts_fbank.jsonl.gz")
```

4.4 CLI usage

An equivalent example using the terminal:

```
lhotse feat write-default-config feat-config.yml
lhotse feat extract-cuts -j 8 -f feat-config.yml \
    data/cuts.jsonl.gz data/cuts_fbank.jsonl.gz data/fbank
```

4.5 Kaldi compatibility caveats

Most of the spectrogram/fbank/mfcc parameters are the same as in Kaldi. However, we are not fully compatible - Kaldi computes energies from a signal scaled between -32,768 to 32,767, while we scale signal between -1.0 and 1.0. It results in Kaldi energies being significantly greater than in Lhotse. Also, by default, we turn off dithering for deterministic feature extraction.

CHAPTER

FIVE

EXECUTING TASKS IN PARALLEL

In this section we will explain how Lhotse uses a generic interface called `Executor` to parallelize some tasks (mostly feature extraction).

There are multiple ways we can parallelize execution of a Python method:

- using multi-threading (single node, single process);
- using multi-processing (single node, multiple processes);
- using distributed processing (multiple nodes, multiple processes).

The `Executor` API, introduced in Python's standard library in `concurrent.futures` module, allows us to use any of these methods, while writing the code independently of how it is going to be parallelized. This module defines two types of *executors*, i.e. `concurrent.futures.ProcessPoolExecutor` and `concurrent.futures.ThreadPoolExecutor`. We refer the reader to the official documentation of concurrent.futures for details. On a high level, these executors accepts tasks in the form of a Python function and an iterable of arguments, and then distribute the tasks among workers, automatically balancing the load (no manual splitting into chunks/batches is necessary).

Some methods in Lhotse (notably: `lhotse.CutSet.compute_and_store_features()`) have a parameter called `executor`, which is set to `None` by default. It means that by default, they are going to run everything in a single thread and process. The user can pass an object satisfying the `Executor` API instead, and these methods will automatically parallelize the underlying tasks.

Multi-processing: This is the recommended way to parallelize the execution for most users. An example of use to extract features on a `lhotse.CutSet`:

```
from concurrent.futures import ProcessPoolExecutor
from lhotse import CutSet, Fbank, LilcomChunkyWriter
num_jobs = 8
with ProcessPoolExecutor(num_jobs) as ex:
    cuts: CutSet = cuts.compute_and_store_features(
        extractor=Fbank(),
        storage=LilcomChunkyWriter('feats'),
        executor=ex
    )
```

Distributed processing: This is the recommended way for more advanced users that have the access and desire to leverage high-performance clusters (e.g. at universities). A library called Dask offers multiple powerful Python interfaces for distributed execution. One of them is called Dask.distributed. It implements the `Executor` API via class `distributed.Client`, that can be connected to an existing Dask cluster. The setup of Dask clusters is beyond the scope of this documentation, however you can find a working implementation for the CLSP Sun Grid Engine system here.

Caution: Dask is an optional dependency for Lhotse and has to be installed separately. You can install it with `pip install dask distributed`.

Multi-threading: We discourage using multi-threading with Python. Python is well known for its issues with multi-threading due to global interpreter lock (GIL), which prohibits most "typical" multi-threaded code from running in parallel. Therefore, usually concurrent tasks have to be executed in separate processes (each with its own interpreter), or use threading at the native (C or C++) level. Lhotse currently does not implement any native components, so we rely on Python-level parallelism.

If you are sure that you want to use multi-threading, you can use `concurrent.futures.ThreadPoolExecutor`. We use it sometimes in Lhotse when we expect the operations to be I/O bound rather than CPU bound (like scanning the filesystem for multiple files).

CHAPTER

SIX

PYTORCH DATASETS

Lhotse supports PyTorch's dataset API, providing implementations for the `Dataset` and `Sampler` concepts. They can be used together with the standard `DataLoader` class for efficient mini-batch collection with multiple parallel readers and pre-fetching.

6.1 A quick re-cap of PyTorch's data API

PyTorch defines the Dataset class that is responsible for reading the data from disk/memory/Internet/database/etc., and converting it to tensors that can be used for network training or inference. These `Dataset`'s are typically „map-style" datasets which are given an index (or a list of indices) and return the corresponding data samples.

The selection of indices is performed by the `Sampler` class. `Sampler`, knowing the length (number of items) in a `Dataset`, can use various strategies to determine the order of elements to read (e.g. sequential reads, or random reads).

More details about the data pipeline API in PyTorch can be found here.

6.2 About Lhotse's Datasets and Samplers

Lhotse provides a number of utilities that make it simpler to define `Dataset`'s for speech processing tasks. *`CutSet`* is the base data structure that is used to initialize the `Dataset` class. This makes it possible to manipulate the speech data in convenient ways - pad, mix, concatenate, augment, compute features, look up the supervision information, etc.

Lhotse's `Dataset`'s will perform batching by themselves, because auto-collation in `DataLoader` is too limiting for speech data handling. These `Dataset`'s expect to be handed lists of element indices, so that they can collate the data *before* it is passed to the `DataLoader` (which must use `batch_size=None`). It allows for interesting collation methods - e.g. **padding the speech with noise recordings, or actual acoustic context**, rather than artificial zeroes; or **dynamic batch sizes**.

The items for mini-batch creation are selected by the `Sampler`. Lhotse defines `Sampler` classes that are initialized with *`CutSet`*'s, so that they can look up specific properties of an utterance to stratify the sampling. For example, `SimpleCutSampler` has a defined `max_frames` attribute, and it will keep sampling cuts for a batch until they do not exceed the specified number of frames. Another strategy — used in `BucketingSampler` — will first group the cuts of similar durations into buckets, and then randomly select a bucket to draw the whole batch from.

For tasks where both input and output of the model are speech utterances, we can use the `CutPairsSampler`, which accepts two *`CutSet`*'s and will match the cuts in them by their IDs.

A typical Lhotse's dataset API usage might look like this:

```python
from torch.utils.data import DataLoader
from lhotse.dataset import SpeechRecognitionDataset, SimpleCutSampler

cuts = CutSet(...)
dset = SpeechRecognitionDataset(cuts)
sampler = SimpleCutSampler(cuts, max_frames=50000)
# Dataset performs batching by itself, so we have to indicate that
# to the DataLoader with batch_size=None
dloader = DataLoader(dset, sampler=sampler, batch_size=None, num_workers=1)
for batch in dloader:
    ...  # process data
```

6.3 Restoring sampler's state: continuing the training

All `CutSampler` types can save their progress and pick up from that checkpoint. For consistency with PyTorch tensors, the relevant methods are called `.state_dict()` and `.load_state_dict()`. The following example illustrates how to save the sampler's state (pay attention to the last bit):

```python
dataset = ...  # Some task-specific dataset initialization
sampler = BucketingSampler(cuts, max_duration=200, shuffle=True, num_buckets=30)
dloader = DataLoader(dataset, batch_size=None, sampler=sampler, num_workers=4)
global_step = 0
for epoch in range(30):
    dloader.sampler.set_epoch(epoch)
    for batch in dloader:
        # ... processing forward, backward, etc.
        global_step += 1

        if global_step % 5000 == 0:
            state = dloader.sampler.state_dict()
            torch.save(state, f'sampler-ckpt-ep{epoch}-step{global_step}.pt')
```

In case that the training is ended abruptly and the epochs are very long (10k+ steps, not uncommon with large datasets these days), we can resume the training from where it left off like the following:

```python
# Creating a vanilla sampler, we will read the previous progress into it.
sampler = BucketingSampler(cuts, max_duration=200, shuffle=True, num_buckets=30)

# Restore the sampler's state.
state = torch.load('sampler-ckpt-ep5-step75000.pt')
sampler.load_state_dict(state)

dloader = DataLoader(dataset, batch_size=None, sampler=sampler, num_workers=4)

global_step = sampler.diagnostics.total_cuts  # <-- Restore the global step idx.
for epoch in range(sampler.epoch, 30):  # <-- Skip previous epochs that are already␣
↪processed.

    dloader.sampler.set_epoch(epoch)
    for batch in dloader:
        # Note: the first batch is going to be from step 75009.
```

(continues on next page)

(continued from previous page)

```
        # With DataLoader num_workers==0, it would have been 75001, but we get
        # +8 because of num_workers==4 * prefetching_factor==2

        # ... processing forward, backward, etc.
        global_step += 1
```

Note: In general, the sampler arguments may be different – loading a `state_dict` will overwrite the arguments, and emit a warning for the user to be aware what happened. `BucketingSampler` is an exception – the `num_buckets` and `bucket_method` must be consistent, otherwise we couldn't guarantee identical outcomes after training resumption.

Note: The `DataLoader`'s `num_workers` can be different after resuming.

6.4 Batch I/O: pre-computed vs. on-the-fly features

Depending on the experimental setup and infrastructure, it might be more convenient to either pre-compute and store features like filter-bank energies for later use (as traditionally done in Kaldi/ESPnet/Espresso toolkits), or compute them dynamically during training ("on-the-fly"). Lhotse supports both modes of computation by introducing a class called *BatchIO*. It is accepted as an argument in most dataset classes, and defaults to *PrecomputedFeatures*. Other available choices are *AudioSamples* for working with waveforms directly, and *OnTheFlyFeatures*, which wraps a *FeatureExtractor* and applies it to a batch of recordings. These strategies automatically pad and collate the inputs, and provide information about the original signal lengths: as a number of frames/samples, binary mask, or start-end frame/sample pairs.

6.4.1 Which strategy to choose?

In general, pre-computed features can be greatly compressed (we achieve 70% size reduction with regard to uncompressed features), and so the I/O load on your computing infrastructure will be much smaller than if you read the recordings directly. This is especially valuable when working with network file systems (NFS) that are typically used in computational grids for storage. When your experiment is I/O bound, then it is best to use pre-computed features.

When I/O is not the issue, it might be preferable to use on-the-fly computation as it shouldn't require any prior steps to perform the network training. It is also simpler to apply a vast range of data augmentation methods in a fully randomized way (e.g. reverberation), although Lhotse provides support for approximate feature-domain signal mixing (e.g. for additive noise augmentation) to alleviate that to some extent.

6.5 Dataset's list

class `lhotse.dataset.diarization.`**`DiarizationDataset`**(*cuts*, *uem=None*, *min_speaker_dim=None*, *global_speaker_ids=False*)

A PyTorch Dataset for the speaker diarization task. Our assumptions about speaker diarization are the following:

- **we assume a single channel input (for now), which could be either a true mono signal** or a beamforming result from a microphone array.

- **we assume that the supervision used for model training is a speech activity matrix, with one** row dedicated to each speaker (either in the current cut or the whole dataset, depending on the settings). The columns correspond to feature frames. Each row is effectively a Voice Activity Detection supervision for a single speaker. This setup is somewhat inspired by the TS-VAD paper: https://arxiv.org/abs/2005.07272

Each item in this dataset is a dict of:

```
{
    'features': (B x T x F) tensor
    'features_lens': (B, ) tensor
    'speaker_activity': (B x num_speaker x T) tensor
}
```

Constructor arguments:

Parameters

- **cuts** (*CutSet*) – a CutSet used to create the dataset object.
- **uem** (Optional[*SupervisionSet*]) – a SupervisionSet used to set regions for diarization
- **min_speaker_dim** (Optional[int]) – optional int, when specified it will enforce that the matrix shape is at least that value (useful for datasets like CHiME 6 where the number of speakers is always 4, but some cuts might have less speakers than that).
- **global_speaker_ids** (bool) – a bool, indicates whether the same speaker should always retain the same row index in the speaker activity matrix (useful for speaker-dependent systems)
- **root_dir** – a prefix path to be attached to the feature files paths.

__init__(*cuts*, *uem=None*, *min_speaker_dim=None*, *global_speaker_ids=False*)

class lhotse.dataset.unsupervised.**UnsupervisedDataset**

Dataset that contains no supervision - it only provides the features extracted from recordings.

```
{
    'features': (B x T x F) tensor
    'features_lens': (B, ) tensor
}
```

__init__()

class lhotse.dataset.unsupervised.**UnsupervisedWaveformDataset**(*collate=True*)

A variant of UnsupervisedDataset that provides waveform samples instead of features. The output is a tensor of shape (C, T), with C being the number of channels and T the number of audio samples. In this implementation, there will always be a single channel.

Returns:

```
{
    'audio': (B x NumSamples) float tensor
    'audio_lens': (B, ) int tensor
}
```

__init__(*collate=True*)

class lhotse.dataset.unsupervised.**DynamicUnsupervisedDataset**(*feature_extractor*, *augment_fn=None*)

An example dataset that shows how to use on-the-fly feature extraction in Lhotse. It accepts two additional inputs - a FeatureExtractor and an optional WavAugmenter for time-domain data augmentation.. The output is approximately the same as that of the `UnsupervisedDataset` - there might be slight differences for `MixedCut``s, because this dataset mixes them in the time domain, and ``UnsupervisedDataset` does that in the feature domain. Cuts that are not mixed will yield identical results in both dataset classes.

__init__(*feature_extractor*, *augment_fn=None*)

class lhotse.dataset.speech_recognition.**K2SpeechRecognitionDataset**(*return_cuts=False*, *cut_transforms=None*, *input_transforms=None*, *input_strategy=<lhotse.dataset.input_strategies.F object>*)

The PyTorch Dataset for the speech recognition task using k2 library.

This dataset expects to be queried with lists of cut IDs, for which it loads features and automatically collates/batches them.

To use it with a PyTorch DataLoader, set `batch_size=None` and provide a `SimpleCutSampler` sampler.

Each item in this dataset is a dict of:

```
{
    'inputs': float tensor with shape determined by :attr:`input_strategy`:
              - single-channel:
                - features: (B, T, F)
                - audio: (B, T)
              - multi-channel: currently not supported
    'supervisions': [
        {
            'sequence_idx': Tensor[int] of shape (S,)
            'text': List[str] of len S

            # For feature input strategies
            'start_frame': Tensor[int] of shape (S,)
            'num_frames': Tensor[int] of shape (S,)

            # For audio input strategies
            'start_sample': Tensor[int] of shape (S,)
            'num_samples': Tensor[int] of shape (S,)

            # Optionally, when return_cuts=True
            'cut': List[AnyCut] of len S
        }
    ]
}
```

Dimension symbols legend: * `B` - batch size (number of Cuts) * `S` - number of supervision segments (greater or equal to B, as each Cut may have multiple supervisions) * `T` - number of frames of the longest Cut * `F` - number of features

The 'sequence_idx' field is the index of the Cut used to create the example in the Dataset.

__init__(*return_cuts=False*, *cut_transforms=None*, *input_transforms=None*, *input_strategy=<lhotse.dataset.input_strategies.PrecomputedFeatures object>*)

k2 ASR IterableDataset constructor.

Parameters

- **return_cuts** (bool) – When True, will additionally return a "cut" field in each batch with the Cut objects used to create that batch.
- **cut_transforms** (Optional[List[Callable[[*CutSet*], *CutSet*]]]) – A list of transforms to be applied on each sampled batch, before converting cuts to an input representation (audio/features). Examples: cut concatenation, noise cuts mixing, etc.
- **input_transforms** (Optional[List[Callable[[Tensor], Tensor]]]) – A list of transforms to be applied on each sampled batch, after the cuts are converted to audio/features. Examples: normalization, SpecAugment, etc.
- **input_strategy** (*BatchIO*) – Converts cuts into a collated batch of audio/features. By default, reads pre-computed features from disk.

lhotse.dataset.speech_recognition.**validate_for_asr**(*cuts*)

Return type None

lhotse.dataset.**speech_synthesis**

alias of <module 'lhotse.dataset.speech_synthesis' from '/home/docs/checkouts/readthedocs.org/user_builds/lhotse/envs/v1.2_a/lib packages/lhotse/dataset/speech_synthesis.py'>

class lhotse.dataset.source_separation.**DynamicallyMixedSourceSeparationDataset**(*sources_set*, *mixtures_set*, *nonsources_set=None*)

A PyTorch Dataset for the source separation task. It's created from a number of CutSets:

- sources_set: provides the audio cuts for the sources that (the targets of source separation),
- mixtures_set: provides the audio cuts for the signal mix (the input of source separation),
- nonsources_set: *(optional)* provides the audio cuts for other signals that are in the mix, but are not the targets of source separation. Useful for adding noise.

When queried for data samples, it returns a dict of:

```
{
    'sources': (N x T x F) tensor,
    'mixture': (T x F) tensor,
    'real_mask': (N x T x F) tensor,
    'binary_mask': (T x F) tensor
}
```

This Dataset performs on-the-fly feature-domain mixing of the sources. It expects the mixtures_set to contain MixedCuts, so that it knows which Cuts should be mixed together.

__init__(*sources_set*, *mixtures_set*, *nonsources_set=None*)

validate()

class lhotse.dataset.source_separation.**PreMixedSourceSeparationDataset**(*sources_set*, *mixtures_set*)

A PyTorch Dataset for the source separation task. It's created from two CutSets - one provides the audio cuts for the sources, and the other one the audio cuts for the signal mix. When queried for data samples, it returns a dict of:

```
{
    'sources': (N x T x F) tensor,
    'mixture': (T x F) tensor,
    'real_mask': (N x T x F) tensor,
    'binary_mask': (T x F) tensor
}
```

It expects both CutSets to return regular Cuts, meaning that the signals were mixed in the time domain. In contrast to DynamicallyMixedSourceSeparationDataset, no on-the-fly feature-domain-mixing is performed.

__init__(*sources_set*, *mixtures_set*)

class lhotse.dataset.vad.**VadDataset**(*input_strategy=<lhotse.dataset.input_strategies.PrecomputedFeatures object>*, *cut_transforms=None*, *input_transforms=None*)

The PyTorch Dataset for the voice activity detection task. Each item in this dataset is a dict of:

```
{
    'inputs': (B x T x F) tensor
    'input_lens': (B,) tensor
    'is_voice': (T x 1) tensor
    'cut': List[Cut]
}
```

__init__(*input_strategy=<lhotse.dataset.input_strategies.PrecomputedFeatures object>*, *cut_transforms=None*, *input_transforms=None*)

6.6 Sampler's list

6.7 Input strategies' list

class lhotse.dataset.input_strategies.**BatchIO**(*num_workers=0*, *executor_type=<class 'concurrent.futures.thread.ThreadPoolExecutor'>*)

Converts a `CutSet` into a collated batch of audio representations. These representations can be e.g. audio samples or features. They might also be single or multi channel.

All InputStrategies support the `executor` parameter in the constructor. It allows to pass a `ThreadPoolExecutor` or a `ProcessPoolExecutor` to parallelize reading audio/features from wherever they are stored. Note that this approach is incompatible with specifying the `num_workers` to `torch.utils.data.DataLoader`, but in some instances may be faster.

Note: This is a base class that only defines the interface.

__call__(*cuts*)

Returns a tensor with collated input signals, and a tensor of length of each signal before padding.

Return type `Tuple[Tensor, IntTensor]`

__init__(*num_workers=0*, *executor_type=<class 'concurrent.futures.thread.ThreadPoolExecutor'>*)

supervision_intervals(*cuts*)

Returns a dict that specifies the start and end bounds for each supervision, as a 1-D int tensor.

Depending on the strategy, the dict should look like:

or

Where S is the total number of supervisions encountered in the CutSet. Note that S might be different than the number of cuts (B). sequence_idx means the index of the corresponding feature matrix (or cut) in a batch.

Return type Dict[str, Tensor]

supervision_masks(*cuts*)
Returns a collated batch of masks, marking the supervised regions in cuts. They are zero-padded to the longest cut.

Depending on the strategy implementation, it is expected to be a tensor of shape (B, NF) or (B, NS), where B denotes the number of cuts, NF the number of frames and NS the total number of samples. NF and NS are determined by the longest cut in a batch.

Return type Tensor

class lhotse.dataset.input_strategies.**PrecomputedFeatures**(*num_workers=0*, *executor_type=<class 'concurrent.futures.thread.ThreadPoolExecutor'>*)
InputStrategy that reads pre-computed features, whose manifests are attached to cuts, from disk.

It automatically pads the feature matrices so that every example has the same number of frames as the longest cut in a mini-batch. This is needed to put all examples into a single tensor. The padding value is a low log-energy, around log(1e-10).

__call__(*cuts*)
Reads the pre-computed features from disk/other storage. The returned shape is (B, T, F) => (batch_size, num_frames, num_features).

Return type Tuple[Tensor, Tensor]

Returns a tensor with collated features, and a tensor of num_frames of each cut before padding.

supervision_intervals(*cuts*)
Returns a dict that specifies the start and end bounds for each supervision, as a 1-D int tensor, in terms of frames:

Where S is the total number of supervisions encountered in the CutSet. Note that S might be different than the number of cuts (B). sequence_idx means the index of the corresponding feature matrix (or cut) in a batch.

Return type Dict[str, Tensor]

supervision_masks(*cuts*, *use_alignment_if_exists=None*)
Returns the mask for supervised frames.

Parameters **use_alignment_if_exists** (Optional[str]) – optional str, key for alignment type to use for generating the mask. If not exists, fall back on supervision time spans.

Return type Tensor

class lhotse.dataset.input_strategies.**AudioSamples**(*num_workers=0*, *fault_tolerant=False*, *executor_type=<class 'concurrent.futures.thread.ThreadPoolExecutor'>*)
InputStrategy that reads single-channel recordings, whose manifests are attached to cuts, from disk (or other audio source).

It automatically zero-pads the recordings so that every example has the same number of audio samples as the longest cut in a mini-batch. This is needed to put all examples into a single tensor.

`__call__`(*cuts*, *recording_field=None*)
Reads the audio samples from recordings on disk/other storage. The returned shape is `(B, T) => (batch_size, num_samples)`.

Return type `Union[Tuple[Tensor, Tensor], Tuple[Tensor, Tensor, CutSet]]`

Returns a tensor with collated audio samples, and a tensor of `num_samples` of each cut before padding.

Parameters `recording_field` (`Optional[str]`) – when specified, we will try to load recordings from a custom field with this name (i.e., `cut.load_<recording_field>()` instead of default `cut.load_audio()`).

`__init__`(*num_workers=0*, *fault_tolerant=False*, *executor_type=<class 'concurrent.futures.thread.ThreadPoolExecutor'>*)
AudioSamples constructor.

Parameters

- `num_workers` (`int`) – when larger than 0, we will spawn an executor (of type specified by `executor_type`) to read the audio data in parallel. Thread executor can be used with PyTorch's DataLoader, whereas Process executor would fail (but could be faster for other applications).
- `fault_tolerant` (`bool`) – when `True`, the cuts for which audio loading failed will be skipped. It will make `__call__` return an additional item, which is the CutSet for which we successfully read the audio. It may be a subset of the input CutSet.
- `executor_type` (`Type[~ExecutorType]`) – the type of executor used for parallel audio reads (only relevant when `num_workers>0`).

`supervision_intervals`(*cuts*)
Returns a dict that specifies the start and end bounds for each supervision, as a 1-D int tensor, in terms of samples:

Where `S` is the total number of supervisions encountered in the `CutSet`. Note that `S` might be different than the number of cuts (`B`). `sequence_idx` means the index of the corresponding feature matrix (or cut) in a batch.

Return type `Dict[str, Tensor]`

`supervision_masks`(*cuts*, *use_alignment_if_exists=None*)
Returns the mask for supervised samples.

Parameters `use_alignment_if_exists` (`Optional[str]`) – optional str, key for alignment type to use for generating the mask. If not exists, fall back on supervision time spans.

Return type `Tensor`

class `lhotse.dataset.input_strategies.OnTheFlyFeatures`(*extractor*, *wave_transforms=None*, *num_workers=0*, *use_batch_extract=True*, *fault_tolerant=False*, *return_audio=False*, *executor_type=<class 'concurrent.futures.thread.ThreadPoolExecutor'>*)
`InputStrategy` that reads single-channel recordings, whose manifests are attached to cuts, from disk (or other audio source). Then, it uses a `FeatureExtractor` to compute their features on-the-fly.

It automatically pads the feature matrices so that every example has the same number of frames as the longest cut in a mini-batch. This is needed to put all examples into a single tensor. The padding value is a low log-energy, around log(1e-10).

`__call__`(*cuts*)
Reads the audio samples from recordings on disk/other storage and computes their features. The returned shape is `(B, T, F) => (batch_size, num_frames, num_features)`.

Return type `Union[Tuple[Tensor, Tensor], Tuple[Tensor, Tensor, CutSet]]`

Returns a tuple of objcets: `(feats, feat_lens, [audios, audio_lens], [cuts])`. Tensors `audios` and `audio_lens` are returned when `return_audio=True`. CutSet `cuts` is returned when `fault_tolerant=True`.

`__init__`(*extractor*, *wave_transforms=None*, *num_workers=0*, *use_batch_extract=True*, *fault_tolerant=False*, *return_audio=False*, *executor_type=<class 'concurrent.futures.thread.ThreadPoolExecutor'>*)
OnTheFlyFeatures' constructor.

Parameters

- **extractor** (`FeatureExtractor`) – the feature extractor used on-the-fly (individually on each waveform).
- **wave_transforms** (`Optional[List[Callable[[Tensor], Tensor]]]`) – an optional list of transforms applied on the batch of audio waveforms collated into a single tensor, right before the feature extraction.
- **num_workers** (`int`) – when larger than 0, we will spawn an executor (of type specified by `executor_type`) to read the audio data in parallel. Thread executor can be used with PyTorch's DataLoader, whereas Process executor would fail (but could be faster for other applications).
- **use_batch_extract** (`bool`) – when `True`, we will call `extract_batch()` to compute the features as it is possibly faster. It has a restriction that all cuts must have the same sampling rate. If that is not the case, set this to `False`.
- **fault_tolerant** (`bool`) – when `True`, the cuts for which audio loading failed will be skipped. It will make `__call__` return an additional item, which is the CutSet for which we successfully read the audio. It may be a subset of the input CutSet.
- **return_audio** (`bool`) – When `True`, calling this object will additionally return collated audio tensor and audio lengths tensor.
- **executor_type** (`Type[~ExecutorType]`) – the type of executor used for parallel audio reads (only relevant when `num_workers>0`).

`supervision_intervals`(*cuts*)
Returns a dict that specifies the start and end bounds for each supervision, as a 1-D int tensor, in terms of frames:

Where `S` is the total number of supervisions encountered in the `CutSet`. Note that `S` might be different than the number of cuts (`B`). `sequence_idx` means the index of the corresponding feature matrix (or cut) in a batch.

Return type `Dict[str, Tensor]`

`supervision_masks`(*cuts*, *use_alignment_if_exists=None*)
Returns the mask for supervised samples.

Parameters **use_alignment_if_exists** (`Optional[str]`) – optional str, key for alignment type to use for generating the mask. If not exists, fall back on supervision time spans.

Return type `Tensor`

6.8 Augmentation - transforms on cuts

Some transforms, in order for us to have accurate information about the start and end times of the signal and its supervisions, have to be performed on cuts (or CutSets).

class lhotse.dataset.cut_transforms.**CutConcatenate**(*gap=1.0*, *duration_factor=1.0*)

A transform on batch of cuts (`CutSet`) that concatenates the cuts to minimize the total amount of padding; e.g. instead of creating a batch with 40 examples, we will merge some of the examples together adding some silence between them to avoid a large number of padding frames that waste the computation.

__init__(*gap=1.0*, *duration_factor=1.0*)

CutConcatenate's constructor.

Parameters

- **gap** (`float`) – The duration of silence in seconds that is inserted between the cuts; it's goal is to let the model "know" that there are separate utterances in a single example.
- **duration_factor** (`float`) – Determines the maximum duration of the concatenated cuts; by default it's 1, setting the limit at the duration of the longest cut in the batch.

class lhotse.dataset.cut_transforms.**CutMix**(*cuts*, *snr=(10, 20)*, *p=0.5*, *prob=None*, *pad_to_longest=True*, *preserve_id=False*)

A transform for batches of cuts (CutSet's) that stochastically performs noise augmentation with a constant or varying SNR.

__init__(*cuts*, *snr=(10, 20)*, *p=0.5*, *prob=None*, *pad_to_longest=True*, *preserve_id=False*)

CutMix's constructor.

Parameters

- **cuts** (*CutSet*) – a `CutSet` containing augmentation data, e.g. noise, music, babble.
- **snr** (`Union[float, Tuple[float, float], None]`) – either a float, a pair (range) of floats, or `None`. It determines the SNR of the speech signal vs the noise signal that's mixed into it. When a range is specified, we will uniformly sample SNR in that range. When it's `None`, the noise will be mixed as-is – i.e. without any level adjustment. Note that it's different from `snr=0`, which will adjust the noise level so that the SNR is 0.
- **prob** (`Optional[float]`) – a float probability in range [0, 1]. Specifies the probability with which we will mix augment the cuts.
- **pad_to_longest** (`bool`) – when *True*, each processed `CutSet` will be padded with noise to match the duration of the longest Cut in a batch.
- **preserve_id** (`bool`) – When `True`, preserves the IDs the cuts had before augmentation. Otherwise, new random IDs are generated for the augmented cuts (default).

class lhotse.dataset.cut_transforms.**ExtraPadding**(*extra_frames=None*, *extra_samples=None*, *extra_seconds=None*, *pad_feat_value=-23.025850929940457*, *randomized=False*, *preserve_id=False*)

A transform on batch of cuts (`CutSet`) that adds a number of extra context frames/samples/seconds on both sides of the cut. Exactly one type of duration has to specified in the constructor.

It is intended mainly for training frame-synchronous ASR models with convolutional layers to avoid using padding inside of the hidden layers, by giving the model larger context in the input. Another useful application is to shift the input by a little, so that the data seen after frame subsampling is a bit different, which makes this a data augmentation technique.

This is best used as the first transform in the transform list for dataset - it will ensure that each individual cut gets extra context before concatenation, or that it will be filled with noise, etc.

__init__(*extra_frames=None*, *extra_samples=None*, *extra_seconds=None*, *pad_feat_value=-23.025850929940457*, *randomized=False*, *preserve_id=False*)

ExtraPadding's constructor.

Parameters

- **extra_frames** (Optional[int]) – The total number of frames to add to each cut. We will add half that number on each side of the cut ("both" directions padding).
- **extra_samples** (Optional[int]) – The total number of samples to add to each cut. We will add half that number on each side of the cut ("both" directions padding).
- **extra_seconds** (Optional[float]) – The total duration in seconds to add to each cut. We will add half that number on each side of the cut ("both" directions padding).
- **pad_feat_value** (float) – When padding a cut with precomputed features, what value should be used for padding (the default is a very low log-energy).
- **randomized** (bool) – When True, we will sample a value from a uniform distribution of [0, extra_X] for each cut (for samples/frames – sample an int, for duration – sample a float).
- **preserve_id** (bool) – When True, preserves the IDs the cuts had before augmentation. Otherwise, new random IDs are generated for the augmented cuts (default).

class lhotse.dataset.cut_transforms.**PerturbSpeed**(*factors*, *p*, *randgen=None*, *preserve_id=False*)

A transform on batch of cuts (CutSet) that perturbs the speed of the recordings with a given probability p.

If the effect is applied, then one of the perturbation factors from the constructor's factors parameter is sampled with uniform probability.

__init__(*factors*, *p*, *randgen=None*, *preserve_id=False*)

class lhotse.dataset.cut_transforms.**PerturbTempo**(*factors*, *p*, *randgen=None*, *preserve_id=False*)

A transform on batch of cuts (CutSet) that perturbs the tempo of the recordings with a given probability p.

If the effect is applied, then one of the perturbation factors from the constructor's factors parameter is sampled with uniform probability.

__init__(*factors*, *p*, *randgen=None*, *preserve_id=False*)

class lhotse.dataset.cut_transforms.**PerturbVolume**(*p*, *scale_low=0.125*, *scale_high=2.0*, *randgen=None*, *preserve_id=False*)

A transform on batch of cuts (CutSet) that perturbs the volume of the recordings with a given probability p.

If the effect is applied, then one of the perturbation factors from the constructor's factors parameter is sampled with uniform probability.

__init__(*p*, *scale_low=0.125*, *scale_high=2.0*, *randgen=None*, *preserve_id=False*)

class lhotse.dataset.cut_transforms.**ReverbWithImpulseResponse**(*rir_recordings*, *p*, *normalize_output=True*, *randgen=None*, *preserve_id=False*, *early_only=False*, *rir_channels=[0]*)

A transform on batch of cuts (CutSet) that convolves each cut with an impulse response with some probability p. The impulse response is chosen randomly from a specified CutSet of RIRs rir_cuts. If *early_only* is set to True, convolution is performed only with the first 50ms of

the impulse response.

`__init__`(*rir_recordings*, *p*, *normalize_output=True*, *randgen=None*, *preserve_id=False*, *early_only=False*, *rir_channels=[0]*)

6.9 Augmentation - transforms on signals

These transforms work directly on batches of collated feature matrices (or possibly raw waveforms, if applicable).

class lhotse.dataset.signal_transforms.**GlobalMVN**(*feature_dim*)
Apply global mean and variance normalization

`__init__`(*feature_dim*)
Initializes internal Module state, shared by both nn.Module and ScriptModule.

classmethod **from_cuts**(*cuts*, *max_cuts=None*)

Return type `GlobalMVN`

classmethod **from_file**(*stats_file*)

Return type `GlobalMVN`

to_file(*stats_file*)

forward(*features*, *supervision_segments=None*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

Return type `Tensor`

inverse(*features*)

Return type `Tensor`

training: bool

class lhotse.dataset.signal_transforms.**SpecAugment**(*time_warp_factor=80*, *num_feature_masks=2*, *features_mask_size=27*, *num_frame_masks=10*, *frames_mask_size=100*, *max_frames_mask_fraction=0.15*, *p=0.9*)
SpecAugment performs three augmentations: - time warping of the feature matrix - masking of ranges of features (frequency bands) - masking of ranges of frames (time)

The current implementation works with batches, but processes each example separately in a loop rather than simultaneously to achieve different augmentation parameters for each example.

`__init__`(*time_warp_factor=80*, *num_feature_masks=2*, *features_mask_size=27*, *num_frame_masks=10*, *frames_mask_size=100*, *max_frames_mask_fraction=0.15*, *p=0.9*)
SpecAugment's constructor.

Parameters

- **time_warp_factor** (Optional[int]) – parameter for the time warping; larger values mean more warping. Set to None, or less than 1, to disable.
- **num_feature_masks** (int) – how many feature masks should be applied. Set to 0 to disable.
- **features_mask_size** (int) – the width of the feature mask (expressed in the number of masked feature bins). This is the F parameter from the SpecAugment paper.
- **num_frame_masks** (int) – the number of masking regions for utterances. Set to 0 to disable.
- **frames_mask_size** (int) – the width of the frame (temporal) masks (expressed in the number of masked frames). This is the T parameter from the SpecAugment paper.
- **max_frames_mask_fraction** (float) – limits the size of the frame (temporal) mask to this value times the length of the utterance (or supervision segment). This is the parameter denoted by p in the SpecAugment paper.
- **p** – the probability of applying this transform. It is different from p in the SpecAugment paper!

forward(*features*, *supervision_segments=None*, **args*, ***kwargs*)

Computes SpecAugment for a batch of feature matrices.

Since the batch will usually already be padded, the user can optionally provide a supervision_segments tensor that will be used to apply SpecAugment only to selected areas of the input. The format of this input is described below.

Parameters

- **features** (Tensor) – a batch of feature matrices with shape (B, T, F).
- **supervision_segments** (Optional[IntTensor]) – an int tensor of shape (S, 3). S is the number of supervision segments that exist in features – there may be either less or more than the batch size. The second dimension encoder three kinds of information: the sequence index of the corresponding feature matrix in *features*, the start frame index, and the number of frames for each segment.

Return type Tensor

Returns an augmented tensor of shape (B, T, F).

state_dict()

Returns a dictionary containing a whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to None are not included.

Returns:

dict: a dictionary containing a whole state of the module

Example:

```
>>> module.state_dict().keys()
['bias', 'weight']
```

Return type Dict

load_state_dict(*state_dict*)

Copies parameters and buffers from *state_dict* into this module and its descendants. If `strict` is `True`, then the keys of *state_dict* must exactly match the keys returned by this module's `state_dict()` function.

Args:

state_dict (dict): a dict containing parameters and persistent buffers.

strict (bool, optional): whether to strictly enforce that the keys in *state_dict* match the keys returned by this module's `state_dict()` function. Default: `True`

Returns:

`NamedTuple` with `missing_keys` and `unexpected_keys` fields:

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

Note: If a parameter or buffer is registered as `None` and its corresponding key exists in *state_dict*, *load_state_dict()* will raise a `RuntimeError`.

training: bool

class lhotse.dataset.signal_transforms.**RandomizedSmoothing**(*sigma=0.1*, *sample_sigma=True*, *p=0.3*)

Randomized smoothing - gaussian noise added to an input waveform, or a batch of waveforms. The summed audio is clipped to `[-1.0, 1.0]` before returning.

__init__(*sigma=0.1*, *sample_sigma=True*, *p=0.3*)

RandomizedSmoothing's constructor.

Parameters

- **sigma** (`Union[float, Sequence[Tuple[int, float]]]`) – standard deviation of the gaussian noise. Either a constant float, or a schedule, i.e. a list of tuples that specify which value to use from which step. For example, `[(0, 0.01), (1000, 0.1)]` means that from steps 0-999, the sigma value will be 0.01, and from step 1000 onwards, it will be 0.1.
- **sample_sigma** (`bool`) – when `False`, then sigma is used as the standard deviation in each forward step. When `True`, the standard deviation is sampled from a uniform distribution of `[-sigma, sigma]` for each forward step.
- **p** (`float`) – the probability of applying this transform.

forward(*audio*, **args*, ***kwargs*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

Return type `Tensor`

training: bool

class lhotse.dataset.signal_transforms.**DereverbWPE**(*n_fft=512*, *hop_length=128*)

Dereverberation with Weighted Prediction Error (WPE). The implementation and default values are borrowed from *nara_wpe* package: https://github.com/fgnt/nara_wpe

The method and library are described in the following paper: https://groups.uni-paderborn.de/nt/pubs/2018/ITG_2018_Drude_Paper.pdf

__init__(*n_fft=512*, *hop_length=128*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*audio*, **args*, ***kwargs*)

Expects audio to be 2D or 3D tensor. 2D means a batch of single-channel audio, shape (B, T). 3D means a batch of multi-channel audio, shape (B, D, T). B => batch size; D => number of channels; T => number of audio samples.

Return type `Tensor`

training: bool

6.10 Collation utilities for building custom Datasets

class lhotse.dataset.collation.**TokenCollater**(*cuts*, *add_eos=True*, *add_bos=True*, *pad_symbol='<pad>'*, *bos_symbol='<bos>'*, *eos_symbol='<eos>'*, *unk_symbol='<unk>'*)

Collate list of tokens

Map sentences to integers. Sentences are padded to equal length. Beginning and end-of-sequence symbols can be added. Call .inverse(tokens_batch, tokens_lens) to reconstruct batch as string sentences.

Example:

```
>>> token_collater = TokenCollater(cuts)
>>> tokens_batch, tokens_lens = token_collater(cuts.subset(first=32))
>>> original_sentences = token_collater.inverse(tokens_batch, tokens_lens)
```

Returns:

tokens_batch: IntTensor of shape (B, L) B: batch dimension, number of input sentences L: length of the longest sentence

tokens_lens: IntTensor of shape (B,) Length of each sentence after adding <eos> and <bos> but before padding.

__init__(*cuts*, *add_eos=True*, *add_bos=True*, *pad_symbol='<pad>'*, *bos_symbol='<bos>'*, *eos_symbol='<eos>'*, *unk_symbol='<unk>'*)

inverse(*tokens_batch*, *tokens_lens*)

Return type `List[str]`

lhotse.dataset.collation.**collate_features**(*cuts*, *pad_direction='right'*, *executor=None*)

Load features for all the cuts and return them as a batch in a torch tensor. The output shape is `(batch, time, features)`. The cuts will be padded with silence if necessary.

Parameters

- **cuts** (*CutSet*) – a `CutSet` used to load the features.
- **pad_direction** (`str`) – where to apply the padding (`right`, `left`, or `both`).

- **executor** (Optional[Executor]) – an instance of ThreadPoolExecutor or ProcessPoolExecutor; when provided, we will use it to read the features concurrently.

Return type Tuple[Tensor, Tensor]

Returns a tuple of tensors (features, features_lens).

lhotse.dataset.collation.**collate_audio**(*cuts*, *pad_direction='right'*, *executor=None*, *fault_tolerant=False*, *recording_field=None*)

Load audio samples for all the cuts and return them as a batch in a torch tensor. The output shape is (batch, time). The cuts will be padded with silence if necessary.

Parameters

- **cuts** (*CutSet*) – a CutSet used to load the audio samples.
- **pad_direction** (str) – where to apply the padding (right, left, or both).
- **executor** (Optional[Executor]) – an instance of ThreadPoolExecutor or ProcessPoolExecutor; when provided, we will use it to read audio concurrently.
- **fault_tolerant** (bool) – when True, the cuts for which audio loading failed will be skipped. Setting this parameter will cause the function to return a 3-tuple, where the third element is a CutSet for which the audio data were sucessfully read.
- **recording_field** (Optional[str]) – when specified, we will try to load recordings from a custom field with this name (i.e., cut.load_<recording_field>() instead of default cut.load_audio()).

Return type Union[Tuple[Tensor, Tensor], Tuple[Tensor, Tensor, *CutSet*]]

Returns a tuple of tensors (audio, audio_lens), or (audio, audio_lens, cuts).

lhotse.dataset.collation.**collate_custom_field**(*cuts*, *field*, *pad_value=None*, *pad_direction='right'*)

Load custom arrays for all the cuts and return them as a batch in a torch tensor. The output shapes are:

- **(batch, d0, d1, d2, ...) for lhotse.array.Array of shape (d0, d1, d2, ...).** Note: all arrays have to be of the same shape, as we expect these represent fixed-size embeddings.
- **(batch, d0, pad_dt, d1, ...) for lhotse.array.TemporalArray of shape** (d0, dt, d1, ...) where dt indicates temporal dimension (variable-sized), and pad_dt indicates temporal dimension after padding (equal-sized for all cuts). We expect these represent temporal data, such as alignments, posteriors, features, etc.
- **(batch,) for anything else, such as int or float: we will simply stack them into** a list and tensorize it.

Note: This function disregards the frame_shift attribute of lhotse.array.TemporalArray when padding; it simply pads all the arrays to the longest one found in the mini-batch. Because of that, the function will work correctly even if the user supplied inconsistent meta-data.

Note: Temporal arrays of integer type that are smaller than torch.int64, will be automatically promoted to torch.int64.

Parameters

- **cuts** (*CutSet*) – a CutSet used to load the features.
- **field** (str) – name of the custom field to be retrieved.

- **pad_value** (Union[None, int, float]) – value to be used for padding the temporal arrays. Ignored for non-temporal array and non-array attributes.
- **pad_direction** (str) – where to apply the padding (right, left, or both).

Return type Union[Tensor, Tuple[Tensor, Tensor]]

Returns a collated data tensor, or a tuple of tensors (collated_data, sequence_lens).

lhotse.dataset.collation.**collate_multi_channel_features**(*cuts*)

Load features for all the cuts and return them as a batch in a torch tensor. The cuts have to be of type MixedCut and their tracks will be interpreted as individual channels. The output shape is (batch, channel, time, features). The cuts will be padded with silence if necessary.

Return type Tensor

lhotse.dataset.collation.**collate_multi_channel_audio**(*cuts*)

Load audio samples for all the cuts and return them as a batch in a torch tensor. The cuts have to be of type MixedCut and their tracks will be interpreted as individual channels. The output shape is (batch, channel, time). The cuts will be padded with silence if necessary.

Return type Tensor

lhotse.dataset.collation.**collate_vectors**(*tensors*, *padding_value=- 100*, *matching_shapes=False*)

Convert an iterable of 1-D tensors (of possibly various lengths) into a single stacked tensor.

Parameters

- **tensors** (Iterable[Union[Tensor, ndarray]]) – an iterable of 1-D tensors.
- **padding_value** (Union[int, float]) – the padding value inserted to make all tensors have the same length.
- **matching_shapes** (bool) – when True, will fail when input tensors have different shapes.

Return type Tensor

Returns a tensor with shape (B, L) where B is the number of input tensors and L is the number of items in the longest tensor.

lhotse.dataset.collation.**collate_matrices**(*tensors*, *padding_value=0*, *matching_shapes=False*)

Convert an iterable of 2-D tensors (of possibly various first dimension, but consistent second dimension) into a single stacked tensor.

Parameters

- **tensors** (Iterable[Union[Tensor, ndarray]]) – an iterable of 2-D tensors.
- **padding_value** (Union[int, float]) – the padding value inserted to make all tensors have the same length.
- **matching_shapes** (bool) – when True, will fail when input tensors have different shapes.

Return type Tensor

Returns a tensor with shape (B, L, F) where B is the number of input tensors, L is the largest found shape[0], and F is equal to shape[1].

lhotse.dataset.collation.**maybe_pad**(*cuts*, *duration=None*, *num_frames=None*, *num_samples=None*, *direction='right'*, *preserve_id=False*)

Check if all cuts' durations are equal and pad them to match the longest cut otherwise.

Return type *CutSet*

lhotse.dataset.collation.**read_audio_from_cuts**(*cuts*, *executor=None*, *suppress_errors=False*, *recording_field=None*)

Loads audio data from an iterable of cuts.

Parameters

- **cuts** (Iterable[*Cut*]) – a CutSet or iterable of cuts.
- **executor** (Optional[Executor]) – optional Executor (e.g., ThreadPoolExecutor or ProcessPoolExecutor) to perform the audio reads in parallel.
- **suppress_errors** (bool) – when set to True, will enable fault-tolerant data reads; we will skip the cuts and audio data for the instances that failed (and emit a warning). When False (default), the errors will not be suppressed.
- **recording_field** (Optional[str]) – when specified, we will try to load recordings from a custom field with this name (i.e., cut.load_<recording_field>() instead of default cut.load_audio()).

Return type Tuple[List[Tensor], *CutSet*]

Returns a tuple of two items: a list of audio tensors (with different shapes), and a list of cuts for which we read the data successfully.

lhotse.dataset.collation.**read_features_from_cuts**(*cuts*, *executor=None*)

Return type List[Tensor]

CHAPTER

SEVEN

KALDI INTEROPERABILITY

7.1 Data import/export

We support importing Kaldi data directories that contain at least the `wav.scp` file, required to create the *RecordingSet*. Other files, such as `segments`, `utt2spk`, etc. are used to create the *SupervisionSet*. We also support converting `feats.scp` to *FeatureSet*, and reading features directly from Kaldi's scp/ark files via kaldi_native_io library (which is an optional Lhotse's dependency).

We also allow to export a pair of *RecordingSet* and *SupervisionSet* to a Kaldi data directory.

We currently do not support the following (but may start doing so in the future):

- Exporting Lhotse extracted features to Kaldi's `feats.scp`
- Export Lhotse's multi-channel recording sets to Kaldi

7.2 Kaldi feature extractors

We support Kaldi-compatible log-mel filter energies ("fbank") and MFCCs. We provide a PyTorch implementation that is GPU-compatible, allows batching, and backpropagation. To learn more about feature extraction in Lhotse, see *Feature extraction*.

7.3 Python

Python methods related to Kaldi support:

`lhotse.kaldi.`**`get_duration`**(*path*)

Read a audio file, it supports pipeline style wave path and real waveform.

Parameters **`path`** (`Union[Path, str]`) – Path to an audio file or a Kaldi-style pipe.

Return type `float`

Returns float duration of the recording, in seconds.

`lhotse.kaldi.`**`load_kaldi_data_dir`**(*path*, *sampling_rate*, *frame_shift=None*, *map_string_to_underscores=None*, *num_jobs=1*)

Load a Kaldi data directory and convert it to a Lhotse RecordingSet and SupervisionSet manifests. For this to work, at least the wav.scp file must exist. SupervisionSet is created only when a segments file exists. All the other files (text, utt2spk, etc.) are optional, and some of them might not be handled yet. In particular, feats.scp files are ignored.

Parameters **map_string_to_underscores** (Optional[str]) – optional string, when specified, we will replace all instances of this string in SupervisonSegment IDs to underscores. This is to help with handling underscores in Kaldi (see *export_to_kaldi()*). This is also done for speaker IDs.

Return type Tuple[*RecordingSet*, Optional[*SupervisionSet*], Optional[*FeatureSet*]]

lhotse.kaldi.**export_to_kaldi**(*recordings*, *supervisions*, *output_dir*, *map_underscores_to=None*, *prefix_spk_id=False*)

Export a pair of RecordingSet and SupervisionSet to a Kaldi data directory. It even supports recordings that have multiple channels but the recordings will still have to have a single AudioSource.

The RecordingSet and SupervisionSet must be compatible, i.e. it must be possible to create a CutSet out of them.

Parameters

- **recordings** (*RecordingSet*) – a RecordingSet manifest.
- **supervisions** (*SupervisionSet*) – a SupervisionSet manifest.
- **output_dir** (Union[Path, str]) – path where the Kaldi-style data directory will be created.
- **map_underscores_to** (Optional[str]) – optional string with which we will replace all underscores. This helps avoid issues with Kaldi data dir sorting.
- **prefix_spk_id** (Optional[bool]) – add speaker_id as a prefix of utterance_id (this is to ensure correct sorting inside files which is required by Kaldi)

lhotse.kaldi.**load_kaldi_text_mapping**(*path*, *must_exist=False*)

Load Kaldi files such as utt2spk, spk2gender, text, etc. as a dict.

Return type Dict[str, Optional[str]]

lhotse.kaldi.**save_kaldi_text_mapping**(*data*, *path*)

Save flat dicts to Kaldi files such as utt2spk, spk2gender, text, etc.

lhotse.kaldi.**make_wavscp_channel_string_map**(*source*, *sampling_rate*)

Return type Dict[int, str]

7.4 CLI

Converting Kaldi data directory called data/train, with 16kHz sampling rate recordings, to a directory with Lhotse manifests called train_manifests:

```
# Convert data/train to train_manifests/{recordings,supervisions}.json
lhotse kaldi import \
    data/train \
    16000 \
    train_manifests

# Convert train_manifests/{recordings,supervisions}.json to data/train
lhotse kaldi export \
    train_manifests/recordings.json \
    train_manifests/supervisions.json \
    data/train
```

CHAPTER

EIGHT

COMMAND-LINE INTERFACE

8.1 lhotse

The shell entry point to Lhotse, a tool and a library for audio data manipulation in high altitudes.

```
lhotse [OPTIONS] COMMAND [ARGS]...
```

Options

-s, --seed <seed>
Random seed.

8.1.1 combine

Load MANIFESTS, combine them into a single one, and write it to OUTPUT_MANIFEST.

```
lhotse combine [OPTIONS] [MANIFESTS]... OUTPUT_MANIFEST
```

Arguments

MANIFESTS
Optional argument(s)

OUTPUT_MANIFEST
Required argument

8.1.2 copy

Load INPUT_MANIFEST and store it to OUTPUT_MANIFEST. Useful for conversion between different serialization formats (e.g. JSON, JSONL, YAML). Automatically supports gzip compression when '.gz' suffix is detected.

```
lhotse copy [OPTIONS] INPUT_MANIFEST OUTPUT_MANIFEST
```

Arguments

INPUT_MANIFEST
Required argument

OUTPUT_MANIFEST
Required argument

8.1.3 copy-feats

Load INPUT_MANIFEST of type `lhotse.FeatureSet` or *lhotse.CutSet*, read every feature matrix using `features.load()` or `cut.load_features()`, save them in STORAGE_PATH and save the updated manifest to OUTPUT_MANIFEST.

```
lhotse copy-feats [OPTIONS] INPUT_MANIFEST OUTPUT_MANIFEST STORAGE_PATH
```

Options

-t, --storage-type <storage_type>
Which storage backend should we use for writing the copied features.

Options chunked_lilcom_hdf5 | kaldiio | lilcom_chunky | lilcom_files | lilcom_hdf5 | lilcom_url | memory_lilcom | memory_raw | numpy_files | numpy_hdf5

-j, --max-jobs <max_jobs>
Maximum number of parallel copying processes. By default, one process is spawned for every existing feature file in the INPUT_MANIFEST (e.g., if the features were extracted with 20 jobs, there will typically be 20 files).

Arguments

INPUT_MANIFEST
Required argument

OUTPUT_MANIFEST
Required argument

STORAGE_PATH
Required argument

8.1.4 cut

Group of commands used to create CutSets.

```
lhotse cut [OPTIONS] COMMAND [ARGS]...
```

append

Create a new CutSet by appending the cuts in CUT_MANIFESTS. CUT_MANIFESTS are iterated position-wise (the cuts on i'th position in each manfiest are appended to each other). The cuts are appended in the order in which they appear in the input argument list. If CUT_MANIFESTS have different lengths, the script stops once the shortest CutSet is depleted.

```
lhotse cut append [OPTIONS] [CUT_MANIFESTS]... OUTPUT_CUT_MANIFEST
```

Arguments

CUT_MANIFESTS
Optional argument(s)

OUTPUT_CUT_MANIFEST
Required argument

decompose

Decompose CUTSET into:

- recording set (recordings.jsonl.gz)
- feature set (features.jsonl.gz)
- supervision set (supervisions.jsonl.gz)

If any of these are not preset in any of the cuts, the corresponding file for them will be empty.

```
lhotse cut decompose [OPTIONS] CUTSET OUTPUT
```

Arguments

CUTSET
Required argument

OUTPUT
Required argument

describe

Describe some statistics of CUTSET, such as the total speech and audio duration.

```
lhotse cut describe [OPTIONS] CUTSET
```

Arguments

CUTSET
Required argument

export-to-webdataset

Export CUTS into a WebDataset tarfile, or a collection of tarfile shards, as specified by WSPECIFIER.

WSPECIFIER can be: - a regular path (e.g., "data/cuts.tar"), - a path template for sharding (e.g., "data/shard-06%d.tar"), or - a "pipe:" expression (e.g., "pipe:gzip -c > data/shard-06%d.tar.gz").

The resulting CutSet contains audio/feature data in addition to metadata, and can be read in Python using 'CutSet.from_webdataset' API.

This function is useful for I/O intensive applications where random reads are too slow, and a one-time lengthy export step that enables fast sequential reading is preferable.

See the WebDataset project for more information: https://github.com/webdataset/webdataset

```
lhotse cut export-to-webdataset [OPTIONS] CUTSET WSPECIFIER
```

Options

-s, --shard-size <shard_size>
Number of cuts per shard (sharding disabled if not defined).

-f, --audio-format <audio_format>
Format in which the audio is encoded (uses torchaudio available formats).

--audio, --no-audio
Should we load and add audio data.

--features, --no-features
Should we load and add feature data.

--custom, --no-custom
Should we load and add custom data.

--fault-tolerant, --stop-on-fail
Should we omit the cuts for which loading data failed, or stop the execution.

Arguments

CUTSET
Required argument

WSPECIFIER
Required argument

mix-by-recording-id

Create a CutSet stored in OUTPUT_CUT_MANIFEST by matching the Cuts from CUT_MANIFESTS by their recording IDs and mixing them together.

```
lhotse cut mix-by-recording-id [OPTIONS] [CUT_MANIFESTS]...
                                OUTPUT_CUT_MANIFEST
```

Arguments

CUT_MANIFESTS
Optional argument(s)

OUTPUT_CUT_MANIFEST
Required argument

mix-sequential

Create a CutSet stored in OUTPUT_CUT_MANIFEST by iterating jointly over CUT_MANIFESTS and mixing the Cuts on the same positions. E.g. the first output cut is created from the first cuts in each input manifest. The mix is performed by summing the features from all Cuts. If the CUT_MANIFESTS have different number of Cuts, the mixing ends when the shorter manifest is depleted.

```
lhotse cut mix-sequential [OPTIONS] [CUT_MANIFESTS]... OUTPUT_CUT_MANIFEST
```

Arguments

CUT_MANIFESTS
Optional argument(s)

OUTPUT_CUT_MANIFEST
Required argument

pad

Create a new CutSet by padding the cuts in CUT_MANIFEST. The cuts will be right-padded, i.e. the padding is placed after the signal ends.

```
lhotse cut pad [OPTIONS] CUT_MANIFEST OUTPUT_CUT_MANIFEST
```

Options

-d, --duration <duration>
Desired duration of cuts after padding. Cuts longer than this won't be affected. By default, pad to the longest cut duration found in CUT_MANIFEST.

Arguments

CUT_MANIFEST
Required argument

OUTPUT_CUT_MANIFEST
Required argument

random-mixed

Create a CutSet stored in OUTPUT_CUT_MANIFEST that contains supervision regions from SUPERVISION_MANIFEST and features supplied by FEATURE_MANIFEST. It first creates a trivial CutSet, splits it into two equal, randomized parts and mixes their features. The parameters of the mix are controlled via SNR_RANGE and OFFSET_RANGE.

```
lhotse cut random-mixed [OPTIONS] SUPERVISION_MANIFEST FEATURE_MANIFEST
                        OUTPUT_CUT_MANIFEST
```

Options

-s, --snr-range <snr_range>
Range of SNR values (in dB) that will be uniformly sampled in order to mix the signals.

-o, --offset-range <offset_range>
Range of relative offset values (0 - 1), which will offset the "right" signal by this many times the duration of the "left" signal. It is uniformly sampled for each mix operation.

Arguments

SUPERVISION_MANIFEST
Required argument

FEATURE_MANIFEST
Required argument

OUTPUT_CUT_MANIFEST
Required argument

simple

Create a CutSet stored in OUTPUT_CUT_MANIFEST. Depending on the provided options, it may contain any combination of recording, feature and supervision manifests. Either RECORDING_MANIFEST or FEATURE_MANIFEST has to be provided. When SUPERVISION_MANIFEST is provided, the cuts time span will correspond to that of the supervision segments. Otherwise, that time span corresponds to the one found in features, if available, otherwise recordings.

```
lhotse cut simple [OPTIONS] OUTPUT_CUT_MANIFEST
```

Options

-r, --recording-manifest <recording_manifest>
Optional recording manifest - will be used to attach the recordings to the cuts.

-f, --feature-manifest <feature_manifest>
Optional feature manifest - will be used to attach the features to the cuts.

-s, --supervision-manifest <supervision_manifest>
Optional supervision manifest - will be used to attach the supervisions to the cuts.

Arguments

OUTPUT_CUT_MANIFEST
Required argument

trim-to-supervisions

Splits each input cut into as many cuts as there are supervisions. These cuts have identical start times and durations as the supervisions. When there are overlapping supervisions, they can be kept or discarded with options.

For example, the following cut:

Cut

|-----------------| Sup1

|----| **Sup2** |-----------|

is transformed into two cuts:

Cut1

|----| Sup1

|----| Sup2 |-|

Cut2

|-----------| Sup1 |-|

Sup2

|-----------|

```
lhotse cut trim-to-supervisions [OPTIONS] CUTS OUTPUT_CUTS
```

Options

--keep-overlapping, --discard-overlapping
when *False*, it will discard parts of other supervisions that overlap with the main supervision. In the illustration, it would discard *Sup2* in *Cut1* and *Sup1* in *Cut2*.

-d, --min-duration <min_duration>
An optional duration in seconds; specifying this argument will extend the cuts that would have been shorter than *min_duration* with actual acoustic context in the recording/features. If there are supervisions present in the context, they are kept when *keep_overlapping* is true. If there is not enough context, the returned cut will be shorter than *min_duration*. If the supervision segment is longer than *min_duration*, the return cut will be longer.

-c, --context-direction <context_direction>
Which direction should the cut be expanded towards to include context. The value of "center" implies equal expansion to left and right; random uniformly samples a value between "left" and "right".

Options center | left | right | random

Arguments

`CUTS`
Required argument

`OUTPUT_CUTS`
Required argument

truncate

Truncate the cuts in the CUT_MANIFEST and write them to OUTPUT_CUT_MANIFEST. Cuts shorter than MAX_DURATION will not be modified.

```
lhotse cut truncate [OPTIONS] CUT_MANIFEST OUTPUT_CUT_MANIFEST
```

Options

--preserve-id
Should the cuts preserve IDs (by default, they will get new, random IDs)

-d, --max-duration <max_duration>
Required The maximum duration in seconds of a cut in the resulting manifest.

-o, --offset-type <offset_type>
Where should the truncated cut start: "start" - at the start of the original cut, "end" - MAX_DURATION before the end of the original cut, "random" - randomly choose somewhere between "start" and "end" options.

Options start | end | random

--keep-overflowing-supervisions, --discard-overflowing-supervisions
When a cut is truncated in the middle of a supervision segment, should the supervision be kept.

Arguments

`CUT_MANIFEST`
Required argument

`OUTPUT_CUT_MANIFEST`
Required argument

windowed

Create a CutSet stored in OUTPUT_CUT_MANIFEST from feature regions in FEATURE_MANIFEST. The feature matrices are traversed in windows with CUT_SHIFT increments, creating cuts of constant CUT_DURATION.

```
lhotse cut windowed [OPTIONS] FEATURE_MANIFEST OUTPUT_CUT_MANIFEST
```

Options

-d, --cut-duration <cut_duration>
How long should the cuts be in seconds.

-s, --cut-shift <cut_shift>
How much to shift the cutting window in seconds (by default the shift is equal to CUT_DURATION).

--keep-shorter-windows, --discard-shorter-windows
When true, the last window will be used to create a Cut even if its duration is shorter than CUT_DURATION.

Arguments

FEATURE_MANIFEST
Required argument

OUTPUT_CUT_MANIFEST
Required argument

8.1.5 download

Command group for download and extract data.

```
lhotse download [OPTIONS] COMMAND [ARGS]...
```

adept

ADEPT prosody transfer evaluation corpus download.

```
lhotse download adept [OPTIONS] TARGET_DIR
```

Arguments

TARGET_DIR
Required argument

aidatatang-200zh

aidatatang_200zh download.

Args:

target_dir: It will create a dir aidatatang_200zh to contain all downloaded/extracted files

```
lhotse download aidatatang-200zh [OPTIONS] TARGET_DIR
```

Arguments

TARGET_DIR
Required argument

aishell

Aishell download.

```
lhotse download aishell [OPTIONS] TARGET_DIR
```

Arguments

TARGET_DIR
Required argument

aishell4

AISHELL-4 download.

```
lhotse download aishell4 [OPTIONS] TARGET_DIR
```

Arguments

TARGET_DIR
Required argument

ali-meeting

AliMeeting download.

```
lhotse download ali-meeting [OPTIONS] TARGET_DIR
```

Options

--force-download

Arguments

TARGET_DIR
Required argument

ami

AMI download.

```
lhotse download ami [OPTIONS] TARGET_DIR
```

Options

--annotations <annotations>
To download annotations in a different directory than corpus.

--mic <mic>
AMI microphone setting.

Options ihm | ihm-mix | sdm | mdm

--url <url>
AMI data downloading URL.

--force-download <force_download>
If True, download even if file is present.

Arguments

TARGET_DIR
Required argument

bvcc

BVCC/VoiceMOS challange data cannot be downloaded.

See info and instructions how to obtain BVCC dataset used for VoiceMOS challange: - https://arxiv.org/abs/2105.02373 - https://nii-yamagishilab.github.io/ecooper-demo/VoiceMOS2022/index.html - https://codalab.lisn.upsaclay.fr/competitions/695

```
lhotse download bvcc [OPTIONS]
```

cmu-arctic

CMU Arctic download.

```
lhotse download cmu-arctic [OPTIONS] TARGET_DIR
```

Arguments

TARGET_DIR
Required argument

cmu-indic

CMU Indic download.

```
lhotse download cmu-indic [OPTIONS] TARGET_DIR
```

Arguments

TARGET_DIR
Required argument

earnings21

Earnings21 dataset download.

```
lhotse download earnings21 [OPTIONS] TARGET_DIR
```

Arguments

TARGET_DIR
Required argument

earnings22

Earnings22 dataset download.

```
lhotse download earnings22 [OPTIONS]
```

gigaspeech

Gigaspeech download.

```
lhotse download gigaspeech [OPTIONS] PASSWORD TARGET_DIR
```

Options

--subset <subset>
Which parts of Gigaspeech to download (by default XL + DEV + TEST).

Options auto | XL | L | M | S | XS | DEV | TEST

--host <host>
Which host to download Gigaspeech.

Arguments

PASSWORD
Required argument

TARGET_DIR
Required argument

heroico

heroico download.

```
lhotse download heroico [OPTIONS] TARGET_DIR
```

Arguments

TARGET_DIR
Required argument

hifitts

HiFiTTS data download.

```
lhotse download hifitts [OPTIONS] TARGET_DIR
```

Arguments

TARGET_DIR
Required argument

icsi

ICSI data download.

```
lhotse download icsi [OPTIONS] AUDIO_DIR
```

Options

--transcripts-dir <transcripts_dir>
To download annotations in a different directory than audio.

--mic <mic>
ICSI microphone setting.

Options ihm | ihm-mix | sdm | mdm

--url <url>
ICSI data downloading URL.

--force-download <force_download>
If True, download even if file is present.

Arguments

AUDIO_DIR
Required argument

libricss

Download LibriCSS dataset.

```
lhotse download libricss [OPTIONS] TARGET_DIR
```

Options

--force-download
Force download

Arguments

TARGET_DIR
Required argument

librimix

Mini LibriMix download.

```
lhotse download librimix [OPTIONS] TARGET_DIR
```

Arguments

TARGET_DIR
Required argument

librispeech

(Mini) Librispeech download.

```
lhotse download librispeech [OPTIONS] TARGET_DIR
```

Options

--full, --mini
Download Librispeech [default] or mini Librispeech.

Arguments

TARGET_DIR
Required argument

libritts

LibriTTS data download.

```
lhotse download libritts [OPTIONS] TARGET_DIR
```

Arguments

TARGET_DIR
Required argument

ljspeech

LJSpeech download.

```
lhotse download ljspeech [OPTIONS] [TARGET_DIR]
```

Arguments

TARGET_DIR
Optional argument

mtedx

MTEDx download.

```
lhotse download mtedx [OPTIONS] TARGET_DIR
```

Options

-l, --lang <lang>
Specify which languages to download, e.g., lhoste download mtedx . -l de -l fr -l es lhoste download mtedx

Arguments

TARGET_DIR
Required argument

musan

MUSAN download.

```
lhotse download musan [OPTIONS] TARGET_DIR
```

Arguments

TARGET_DIR
Required argument

rir-noise

RIRS and noises download.

```
lhotse download rir-noise [OPTIONS] TARGET_DIR
```

Arguments

TARGET_DIR
Required argument

spgispeech

SPGISpeech download.

```
lhotse download spgispeech [OPTIONS] TARGET_DIR
```

Arguments

TARGET_DIR
Required argument

tedlium

TED-LIUM v3 download (approx. 11GB).

```
lhotse download tedlium [OPTIONS] TARGET_DIR
```

Arguments

TARGET_DIR
Required argument

timit

TIMIT download.

```
lhotse download timit [OPTIONS] TARGET_DIR
```

Arguments

TARGET_DIR
Required argument

vctk

VCTK download.

```
lhotse download vctk [OPTIONS] TARGET_DIR
```

Arguments

TARGET_DIR
Required argument

voxceleb1

VoxCeleb1 download.

```
lhotse download voxceleb1 [OPTIONS] TARGET_DIR
```

Options

--force-download
Force download

Arguments

TARGET_DIR
Required argument

voxceleb2

VoxCeleb2 download.

```
lhotse download voxceleb2 [OPTIONS] TARGET_DIR
```

Options

--force-download
Force download

Arguments

TARGET_DIR
Required argument

yesno

yes_no dataset download.

```
lhotse download yesno [OPTIONS] TARGET_DIR
```

Arguments

TARGET_DIR
Required argument

8.1.6 feat

Feature extraction related commands.

```
lhotse feat [OPTIONS] COMMAND [ARGS]...
```

extract

Extract features for recordings in a given AUDIO_MANIFEST. The features are stored in OUTPUT_DIR, with one file per recording (or segment).

```
lhotse feat extract [OPTIONS] RECORDING_MANIFEST OUTPUT_DIR
```

Options

-f, --feature-manifest <feature_manifest>
Optional manifest specifying feature extractor configuration.

--storage-type <storage_type>
Select a storage backend for the feature matrices.

Options chunked_lilcom_hdf5 | kaldiio | lilcom_chunky | lilcom_files | lilcom_hdf5 | lilcom_url | memory_lilcom | memory_raw | numpy_files | numpy_hdf5

-t, --lilcom-tick-power <lilcom_tick_power>
Determines the compression accuracy; the input will be compressed to integer multiples of 2^tick_power

-r, --root-dir <root_dir>
Root directory - all paths in the manifest will use this as prefix.

-j, --num-jobs <num_jobs>
Number of parallel processes.

Arguments

RECORDING_MANIFEST
Required argument

OUTPUT_DIR
Required argument

extract-cuts

Extract features for cuts in a given CUTSET manifest. The features are stored in STORAGE_PATH, and the output manifest with features is stored in OUTPUT_CUTSET.

```
lhotse feat extract-cuts [OPTIONS] CUTSET OUTPUT_CUTSET STORAGE_PATH
```

Options

-f, --feature-manifest `<feature_manifest>`
Optional manifest specifying feature extractor configuration.

--storage-type `<storage_type>`
Select a storage backend for the feature matrices.

Options chunked_lilcom_hdf5 | kaldiio | lilcom_chunky | lilcom_files | lilcom_hdf5 | lilcom_url | memory_lilcom | memory_raw | numpy_files | numpy_hdf5

-j, --num-jobs `<num_jobs>`
Number of parallel processes.

Arguments

CUTSET
Required argument

OUTPUT_CUTSET
Required argument

STORAGE_PATH
Required argument

extract-cuts-batch

Extract features for cuts in a given CUTSET manifest. The features are stored in STORAGE_PATH, and the output manifest with features is stored in OUTPUT_CUTSET.

This version enables CUDA acceleration for feature extractors that support it (e.g., kaldifeat extractors).

Example usage of kaldifeat fbank with CUDA:

$ pip install kaldifeat # note: ensure it's compiled with CUDA

$ lhotse feat write-default-config -f kaldifeat-fbank feat.yml

$ sed 's/device: cpu/device: cuda/' feat.yml feat-cuda.yml

$ lhotse feat extract-cuts-batch -f feat-cuda.yml cuts.jsonl cuts_with_feats.jsonl feats.h5

```
lhotse feat extract-cuts-batch [OPTIONS] CUTSET OUTPUT_CUTSET STORAGE_PATH
```

Options

-f, --feature-manifest <feature_manifest>
Optional manifest specifying feature extractor configuration. If you want to use CUDA, you should specify the device in this config.

--storage-type <storage_type>
Select a storage backend for the feature matrices.

Options chunked_lilcom_hdf5 | kaldiio | lilcom_chunky | lilcom_files | lilcom_hdf5 | lilcom_url | memory_lilcom | memory_raw | numpy_files | numpy_hdf5

-j, --num-jobs <num_jobs>
Number of dataloader workers.

-b, --batch-duration <batch_duration>
At most this many seconds of audio will be processed in each batch.

Arguments

CUTSET
Required argument

OUTPUT_CUTSET
Required argument

STORAGE_PATH
Required argument

upload

Read an existing FEATURE_MANIFEST, upload the feature matrices it contains to a URL location, and save a new feature OUTPUT_MANIFEST that refers to the uploaded features.

The URL can refer to endpoints such as AWS S3, GCP, Azure, etc. For example: "s3://my-bucket/my-features" is a valid URL.

This script does not currently support credentials, and assumes that you have the write permissions.

```
lhotse feat upload [OPTIONS] FEATURE_MANIFEST URL OUTPUT_MANIFEST
```

Options

-j, --num-jobs <num_jobs>

Arguments

FEATURE_MANIFEST
Required argument

URL
Required argument

OUTPUT_MANIFEST
Required argument

write-default-config

Save a default feature extraction config to OUTPUT_CONFIG.

```
lhotse feat write-default-config [OPTIONS] OUTPUT_CONFIG
```

Options

-f, --feature-type <feature_type>
Which feature extractor type to use.

Options fbank | kaldi-fbank | kaldi-mfcc | kaldifeat-fbank | kaldifeat-mfcc | librosa-fbank | mfcc | opensmile-extractor | spectrogram

Arguments

OUTPUT_CONFIG
Required argument

8.1.7 filter

Filter a MANIFEST according to the rule specified in PREDICATE, and save the result to OUTPUT_MANIFEST. It is intended to work generically with most manifest types - it supports RecordingSet, SupervisionSet and CutSet.

The PREDICATE specifies which attribute is used for item selection. Some examples: lhotse filter 'duration>4.5' supervision.json output.json lhotse filter 'num_frames<600' cuts.json output.json lhotse filter 'start=0' cuts.json output.json lhotse filter 'channel!=0' audio.json output.json

It currently only supports comparison of numerical manifest item attributes, such as: start, duration, end, channel, num_frames, num_features, etc.

```
lhotse filter [OPTIONS] PREDICATE MANIFEST OUTPUT_MANIFEST
```

Arguments

PREDICATE
Required argument

MANIFEST
Required argument

OUTPUT_MANIFEST
Required argument

8.1.8 fix

Fix a pair of Lhotse RECORDINGS and SUPERVISIONS manifests. It removes supervisions without corresponding recordings and vice versa, trims the supervisions that exceed the recording, etc. Stores the output files in OUTPUT_DIR under the same names as the input files.

```
lhotse fix [OPTIONS] RECORDINGS SUPERVISIONS OUTPUT_DIR
```

Arguments

RECORDINGS
Required argument

SUPERVISIONS
Required argument

OUTPUT_DIR
Required argument

8.1.9 install-sph2pipe

Install the sph2pipe program to handle sphere (.sph) audio files with "shorten" codec compression (needed for older LDC data).

It downloads an archive and then decompresses and compiles the contents.

```
lhotse install-sph2pipe [OPTIONS]
```

Options

--install-dir <install_dir>
Directory where sph2pipe will be downloaded and installed.

--url <url>
URL from which to download sph2pipe.

8.1.10 kaldi

Kaldi import/export related commands.

```
lhotse kaldi [OPTIONS] COMMAND [ARGS]...
```

export

Convert a pair of `RecordingSet` and `SupervisionSet` manifests into a Kaldi-style data directory.

```
lhotse kaldi export [OPTIONS] RECORDINGS SUPERVISIONS OUTPUT_DIR
```

Options

-u, --map-underscores-to <map_underscores_to>
Optional string with which we will replace all underscores.This helps avoid issues with Kaldi data dir sorting.

-p, --prefix-spk-id
Prefix utterance ids with speaker ids.This helps avoid issues with Kaldi data dir sorting.

Arguments

RECORDINGS
Required argument

SUPERVISIONS
Required argument

OUTPUT_DIR
Required argument

import

Convert a Kaldi data dir DATA_DIR into a directory MANIFEST_DIR of lhotse manifests. Ignores feats.scp. The SAMPLING_RATE has to be explicitly specified as it is not available to read from DATA_DIR.

```
lhotse kaldi import [OPTIONS] DATA_DIR SAMPLING_RATE MANIFEST_DIR
```

Options

-f, --frame-shift <frame_shift>
Frame shift (in seconds) is required to support reading feats.scp.

-u, --map-string-to-underscores <map_string_to_underscores>
When specified, we will replace all instances of this string in SupervisonSegment IDs to underscores. This is to help with handling underscores in Kaldi (see 'export_to_kaldi').

-j, --num-jobs <num_jobs>
Number of jobs for computing recording durations.

Arguments

DATA_DIR
Required argument

SAMPLING_RATE
Required argument

MANIFEST_DIR
Required argument

8.1.11 prepare

Command group with data preparation recipes.

```
lhotse prepare [OPTIONS] COMMAND [ARGS]...
```

adept

ADEPT prosody transfer evaluation corpus data preparation.

```
lhotse prepare adept [OPTIONS] CORPUS_DIR OUTPUT_DIR
```

Arguments

CORPUS_DIR
Required argument

OUTPUT_DIR
Required argument

aidatatang-200zh

aidatatang_200zh ASR data preparation.

Args:

corpus_dir: It should contain a subdirectory "aidatatang_200zh"

output_dir: The output directory.

```
lhotse prepare aidatatang-200zh [OPTIONS] CORPUS_DIR OUTPUT_DIR
```

Arguments

CORPUS_DIR
Required argument

OUTPUT_DIR
Required argument

aishell

Aishell ASR data preparation.

```
lhotse prepare aishell [OPTIONS] CORPUS_DIR OUTPUT_DIR
```

Arguments

CORPUS_DIR
Required argument

OUTPUT_DIR
Required argument

aishell4

AISHELL-4 data preparation.

```
lhotse prepare aishell4 [OPTIONS] CORPUS_DIR OUTPUT_DIR
```

Arguments

CORPUS_DIR
Required argument

OUTPUT_DIR
Required argument

ali-meeting

AliMeeting data preparation.

```
lhotse prepare ali-meeting [OPTIONS] CORPUS_DIR OUTPUT_DIR
```

Options

--mic <mic>

Options near | far

Arguments

CORPUS_DIR
Required argument

OUTPUT_DIR
Required argument

ami

AMI data preparation.

```
lhotse prepare ami [OPTIONS] CORPUS_DIR OUTPUT_DIR
```

Options

--annotations <annotations>
Provide if annotations are download in a different directory than corpus.

--mic <mic>
AMI microphone setting.

Options ihm | ihm-mix | sdm | mdm

--partition <partition>
Data partition to use (see http://groups.inf.ed.ac.uk/ami/corpus/datasets.shtml).

Options scenario-only | full-corpus | full-corpus-asr

--normalize-text <normalize_text>
Type of text normalization to apply (kaldi style, by default)

Options none | upper | kaldi

Arguments

CORPUS_DIR
Required argument

OUTPUT_DIR
Required argument

aspire

ASpIRE data preparation.

```
lhotse prepare aspire [OPTIONS] CORPUS_DIR OUTPUT_DIR
```

Options

--mic <mic>

Options single | multi

Arguments

CORPUS_DIR
Required argument

OUTPUT_DIR
Required argument

babel

This is a data preparation recipe for the IARPA BABEL corpus (see: https://www.iarpa.gov/index.php/research-programs/babel). It should support all of the languages available in BABEL. It will prepare the data from the "conversational" part of BABEL.

This script should be invoked separately for each language you want to prepare, e.g.: $ lhotse prepare babel /export/corpora5/Babel/IARPA_BABEL_BP_101 data/cantonese $ lhotse prepare babel /export/corpora5/Babel/BABEL_OP1_103 data/bengali

```
lhotse prepare babel [OPTIONS] CORPUS_DIR OUTPUT_DIR
```

Arguments

CORPUS_DIR
Required argument

OUTPUT_DIR
Required argument

broadcast-news

English Broadcast News 1997 data preparation. It will output three manifests: for recordings, topic sections, and speech segments. It supports the following LDC distributions:

* 1997 English Broadcast News Train (HUB4)

    Speech LDC98S71 Transcripts LDC98T28

This data is not available for free - your institution needs to have an LDC subscription.

```
lhotse prepare broadcast-news [OPTIONS] AUDIO_DIR TRANSCRIPT_DIR OUTPUT_DIR
```

Arguments

AUDIO_DIR
Required argument

TRANSCRIPT_DIR
Required argument

OUTPUT_DIR
Required argument

bvcc

BVCC data preparation.

CORPUS_DIR should contain the following dir structure

./phase1-main/README ./phase1-main/DATA/sets/* ./phase1-main/DATA/wav/* ...

./phase1-ood/README ./phase1-ood/DATA/sets/ ./phase1-ood/DATA/wav/ ...

Check the READMEs for details.

See 'lhotse download bvcc' for links to instructions how to obtain the corpus.

```
lhotse prepare bvcc [OPTIONS] CORPUS_DIR OUTPUT_DIR
```

Options

-nj, --num_jobs <num_jobs>

Arguments

CORPUS_DIR
Required argument

OUTPUT_DIR
Required argument

callhome-egyptian

About the Callhome Egyptian Arabic Corpus

The CALLHOME Egyptian Arabic corpus of telephone speech consists of 120 unscripted telephone conversations between native speakers of Egyptian Colloquial Arabic (ECA), the spoken variety of Arabic found in Egypt. The dialect of ECA that this dictionary represents is Cairene Arabic.

This recipe uses the speech and transcripts available through LDC. In addition, an Egyptian arabic phonetic lexicon (available via LDC) is used to get word to phoneme mappings for the vocabulary. This datasets are:

Speech : LDC97S45 Transcripts : LDC97T19 Lexicon : LDC99L22 (unused here)

To actually read the audio, you will need the SPH2PIPE binary: you can provide its path, so that we will add it in the manifests (otherwise you might need to modify your PATH environment variable to find sph2pipe).

```
lhotse prepare callhome-egyptian [OPTIONS] AUDIO_DIR TRANSCRIPT_DIR OUTPUT_DIR
```

Options

--absolute-paths <absolute_paths>
Whether to return absolute or relative (to the corpus dir) paths for recordings.

Arguments

AUDIO_DIR
Required argument

TRANSCRIPT_DIR
Required argument

OUTPUT_DIR
Required argument

callhome-english

CallHome American English corpus preparation.

Depending on the value of transcript_dir, will prepare either

- if transcript_dir = None, the SRE task (expected corpus `LDC2001S97`).

The setup will reflect speaker diarization on a portion of CALLHOME used in the 2000 NIST speaker recognition evaluation. The 2000 NIST SRE is required, and has an LDC catalog number LDC2001S97. The data is not available for free, but can be licensed from the LDC (Linguistic Data Consortium) * otherwise data for ASR task (expected LDC corpora `LDC97S42` and `LDC97T14`) will be prepared. The data is not available for free, but can be licensed from the LDC (Linguistic Data Consortium)

The data should be located at AUDIO_DIR. Optionally, for the SRE task, RTTM_DIR can be provided that has the contents of http://www.openslr.org/resources/10/; otherwise, we will download it.

To actually read the audio, you will need the SPH2PIPE binary: you can provide its path, so that we will add it in the manifests (otherwise you might need to modify your PATH environment variable to find sph2pipe).

Example:

lhotse prepare callhome-english /export/corpora5/LDC/LDC97S42 –transcript-dir /export/corpora5/LDC/LDC97T14 ./callhome_asr

or

lhotse prepare callhome-english /export/corpora5/LDC/LDC2001S97 ./callhome_sre

```
lhotse prepare callhome-english [OPTIONS] AUDIO_DIR OUTPUT_DIR
```

Options

--rttm-dir <rttm_dir>

--absolute-paths <absolute_paths>
Whether to return absolute or relative (to the corpus dir) paths for recordings.

--transcript-dir <transcript_dir>
Path to the LDC97T14 corpus. Please note that providing this path, the ASR corpus will be prepared, not the SRE corpus!

Arguments

AUDIO_DIR
Required argument

OUTPUT_DIR
Required argument

cmu-arctic

CMU Arctic data preparation.

```
lhotse prepare cmu-arctic [OPTIONS] CORPUS_DIR OUTPUT_DIR
```

Arguments

CORPUS_DIR
Required argument

OUTPUT_DIR
Required argument

cmu-indic

CMU Indic data preparation.

```
lhotse prepare cmu-indic [OPTIONS] CORPUS_DIR OUTPUT_DIR
```

Arguments

CORPUS_DIR
Required argument

OUTPUT_DIR
Required argument

cmu-kids

CMU Kids corpus data preparation.

```
lhotse prepare cmu-kids [OPTIONS] CORPUS_DIR OUTPUT_DIR
```

Options

--absolute-paths <absolute_paths>
Use absolute paths for recordings

Arguments

CORPUS_DIR
Required argument

OUTPUT_DIR
Required argument

commonvoice

Mozilla CommonVoice manifest preparation script. CORPUS_DIR is expected to contain sub-directories that are named with CommonVoice language codes, e.g., “en”, “pl”, etc.

```
lhotse prepare commonvoice [OPTIONS] CORPUS_DIR OUTPUT_DIR
```

Options

-l, --language <language>
Languages to prepare (scans CORPUS_DIR for language codes by default).

-s, --split <split>
Splits to prepare (available options: train, dev, test, validated, invalidated, other)

-j, --num-jobs <num_jobs>
How many threads to use (can give good speed-ups with slow disks).

Arguments

CORPUS_DIR
Required argument

OUTPUT_DIR
Required argument

cslu-kids

CSLU Kids corpus data preparation.

```
lhotse prepare cslu-kids [OPTIONS] CORPUS_DIR OUTPUT_DIR
```

Options

--absolute-paths <absolute_paths>
Use absolute paths for recordings

--normalize-text <normalize_text>
Remove noise tags (<bn>, <bs>) from spontaneous speech transcripts

Arguments

CORPUS_DIR
Required argument

OUTPUT_DIR
Required argument

dihard3

DIHARD3 data preparation.

```
lhotse prepare dihard3 [OPTIONS] OUTPUT_DIR
```

Options

--dev <dev>

--eval <eval>

--uem, --no-uem
Specify whether or not to create UEM supervision

-j, --num-jobs <num_jobs>
Number of jobs to scan corpus directory for recordings.

Arguments

OUTPUT_DIR
Required argument

earnings21

Earnings21 data preparation.

```
lhotse prepare earnings21 [OPTIONS] CORPUS_DIR OUTPUT_DIR
```

Options

--normalize-text, --no-normalize-text
Normalize the text.

Arguments

CORPUS_DIR
Required argument

OUTPUT_DIR
Required argument

earnings22

Earnings22 data preparation.

```
lhotse prepare earnings22 [OPTIONS] CORPUS_DIR OUTPUT_DIR
```

Options

--normalize-text, --no-normalize-text
Normalize the text.

Arguments

CORPUS_DIR
Required argument

OUTPUT_DIR
Required argument

eval2000

The Eval2000 corpus preparation.

This is conversational telephone speech collected as 2-channel, 8kHz-sampled data. The catalog number LDC2002S09 for audio corpora and LDC2002T43 for transcripts.

This data is not available for free - your institution needs to have an LDC subscription.

```
lhotse prepare eval2000 [OPTIONS] CORPUS_DIR OUTPUT_DIR
```

Options

--absolute-paths <absolute_paths>
Whether to return absolute or relative (to the corpus dir) paths for recordings.

Arguments

CORPUS_DIR
Required argument

OUTPUT_DIR
Required argument

fisher-english

The Fisher English Part 1, 2 corpus preparation.

This is conversational telephone speech collected as 2-channel, 8kHz-sampled data. The catalog number LDC2004S13 and LDC2005S13 for audio corpora and LDC2004T19 LDC2005T19 for transcripts.

This data is not available for free - your institution needs to have an LDC subscription.

```
lhotse prepare fisher-english [OPTIONS] CORPUS_DIR OUTPUT_DIR
```

Options

-ad, --audio-dirs <audio_dirs>
Audio dirs, e.g., *LDC2004S13 LDC2005S13*. Multiple corpora can be provided by repeating *-ad*.

-td, --transcript-dirs <transcript_dirs>
Transcript dirs, e.g., *LDC2004T19 LDC2005T19*. Multiple corpora can be provided by repeating *-ad*.

--absolute-paths <absolute_paths>
Whether to return absolute or relative (to the corpus dir) paths for recordings.

-j, --num-jobs <num_jobs>
Number of concurrent processes scanning the audio files.

Arguments

CORPUS_DIR
Required argument

OUTPUT_DIR
Required argument

fisher-spanish

The Fisher Spanish corpus preparation.

This is conversational telephone speech collected as 2-channel -law, 8kHz-sampled data. The catalog number LDC2010S01 for audio corpus and LDC2010T04 for transcripts.

This data is not available for free - your institution needs to have an LDC subscription.

```
lhotse prepare fisher-spanish [OPTIONS] AUDIO_DIR TRANSCRIPT_DIR OUTPUT_DIR
```

Options

--absolute-paths <absolute_paths>
Whether to return absolute or relative (to the corpus dir) paths for recordings.

Arguments

AUDIO_DIR
Required argument

TRANSCRIPT_DIR
Required argument

OUTPUT_DIR
Required argument

gale-arabic

GALE Arabic Phases 1 to 4 Broadcast news and conversation data preparation.

```
lhotse prepare gale-arabic [OPTIONS] OUTPUT_DIR
```

Options

-s, --audio <audio>
Paths to audio dirs, e.g., LDC2013S02. Multiple corpora can be provided by repeating *-s*.

-t, --transcript <transcript>
Paths to transcript dirs, e.g., LDC2013T17. Multiple corpora can be provided by repeating *-t*

--absolute-paths <absolute_paths>
Use absolute paths for recordings

Arguments

OUTPUT_DIR
Required argument

gale-mandarin

GALE Mandarin Broadcast speech data preparation.

```
lhotse prepare gale-mandarin [OPTIONS] OUTPUT_DIR
```

Options

-s, --audio <audio>
Paths to audio dirs, e.g., LDC2013S08. Multiple corpora can be provided by repeating *-s*.

-t, --transcript <transcript>
Paths to transcript dirs, e.g., LDC2013T20. Multiple corpora can be provided by repeating *-t*

--absolute-paths <absolute_paths>
Use absolute paths for recordings

--segment-words <segment_words>
Use 'jieba' package to perform word segmentation on the text

Arguments

OUTPUT_DIR
Required argument

gigaspeech

Gigaspeech ASR data preparation.

```
lhotse prepare gigaspeech [OPTIONS] CORPUS_DIR OUTPUT_DIR
```

Options

--subset <subset>
Which parts of Gigaspeech to download (by default XL + DEV + TEST).

Options auto | XL | L | M | S | XS | DEV | TEST

-j, --num-jobs <num_jobs>
How many threads to use (can give good speed-ups with slow disks).

Arguments

CORPUS_DIR
Required argument

OUTPUT_DIR
Required argument

heroico

heroico Answers ASR data preparation.

```
lhotse prepare heroico [OPTIONS] SPEECH_DIR TRANSCRIPT_DIR OUTPUT_DIR
```

Arguments

SPEECH_DIR
Required argument

TRANSCRIPT_DIR
Required argument

OUTPUT_DIR
Required argument

hifitts

HiFiTTS data preparation.

```
lhotse prepare hifitts [OPTIONS] CORPUS_DIR OUTPUT_DIR
```

Options

-j, --num-jobs <num_jobs>
How many jobs to use (can give good speed-ups with slow disks).

Arguments

CORPUS_DIR
Required argument

OUTPUT_DIR
Required argument

icsi

AMI data preparation.

```
lhotse prepare icsi [OPTIONS] AUDIO_DIR OUTPUT_DIR
```

Options

--transcripts-dir <transcripts_dir>

--mic <mic>
ICSI microphone setting.

Options ihm | ihm-mix | sdm | mdm

--normalize-text
If set, convert all text annotations to upper case (similar to Kaldi)

Arguments

AUDIO_DIR
Required argument

OUTPUT_DIR
Required argument

l2-arctic

L2 Arctic data preparation.

```
lhotse prepare l2-arctic [OPTIONS] CORPUS_DIR OUTPUT_DIR
```

Arguments

CORPUS_DIR
Required argument

OUTPUT_DIR
Required argument

libricss

LibriCSS recording and supervision manifest preparation.

```
lhotse prepare libricss [OPTIONS] CORPUS_DIR OUTPUT_DIR
```

Options

--type <type>
Type of the corpus to prepare

Options ihm | ihm-mix | mdm

Arguments

CORPUS_DIR
Required argument

OUTPUT_DIR
Required argument

librimix

LibrMix source separation data preparation.

```
lhotse prepare librimix [OPTIONS] LIBRIMIX_CSV OUTPUT_DIR
```

Options

--sampling-rate <sampling_rate>
Sampling rate to set in the RecordingSet manifest.

--min-segment-seconds <min_segment_seconds>
Remove segments shorter than MIN_SEGMENT_SECONDS.

--with-precomputed-mixtures, --no-precomputed-mixtures
Optionally create an RecordingSet manifest including the precomputed LibriMix mixtures.

Arguments

LIBRIMIX_CSV
Required argument

OUTPUT_DIR
Required argument

librispeech

(Mini) Librispeech ASR data preparation.

```
lhotse prepare librispeech [OPTIONS] CORPUS_DIR OUTPUT_DIR
```

Options

-p, --dataset-parts <dataset_parts>
List of dataset parts to prepare. To prepare multiple parts, pass each with *-p* Example: *-p train-clean-360 -p dev-other*

-j, --num-jobs <num_jobs>
How many threads to use (can give good speed-ups with slow disks).

Arguments

CORPUS_DIR
Required argument

OUTPUT_DIR
Required argument

libritts

LibriTTs data preparation.

```
lhotse prepare libritts [OPTIONS] CORPUS_DIR OUTPUT_DIR
```

Options

-j, --num-jobs <num_jobs>
How many jobs to use (can give good speed-ups with slow disks).

--link-previous-utterance, --no-previous-utterance
If true adds previous utterance id to supervisions. Useful for reconstructing chains of utterances as they were read from LibriVox books. If previous utterance was skipped from LibriTTS datasets previous_utt label is None. 66% of utterances have previous utterance.

Arguments

CORPUS_DIR
Required argument

OUTPUT_DIR
Required argument

ljspeech

LJSpeech data preparation.

```
lhotse prepare ljspeech [OPTIONS] CORPUS_DIR OUTPUT_DIR
```

Arguments

CORPUS_DIR
Required argument

OUTPUT_DIR
Required argument

mgb2

mgb2 ASR data preparation.

```
lhotse prepare mgb2 [OPTIONS] CORPUS_DIR OUTPUT_DIR
```

Options

--text-cleaning, --no-text-cleaning
Basic text cleaning.

--buck-walter, --no-buck-walter
Use BuckWalter transliteration.

-j, --num-jobs <num_jobs>
How many threads to use (can give good speed-ups with slow disks).

--mer-thresh <mer_thresh>
filter out segments based on mer (Match Error Rate).

Arguments

CORPUS_DIR
Required argument

OUTPUT_DIR
Required argument

mls

Multilingual Librispeech (MLS) data preparation.

Multilingual LibriSpeech (MLS) dataset is a large multilingual corpus suitable for speech research. The dataset is derived from read audiobooks from LibriVox and consists of 8 languages - English, German, Dutch, Spanish, French, Italian, Portuguese, Polish. It is available at OpenSLR: http://openslr.org/94

```
lhotse prepare mls [OPTIONS] CORPUS_DIR OUTPUT_DIR
```

Options

--opus, --flac
Which codec should be used (OPUS or FLAC)

-j, --num-jobs <num_jobs>
How many threads to use (can give good speed-ups with slow disks).

Arguments

CORPUS_DIR
Required argument

OUTPUT_DIR
Required argument

mtedx

MTEDx ASR data preparation.

```
lhotse prepare mtedx [OPTIONS] CORPUS_DIR OUTPUT_DIR
```

Options

-j, --num-jobs <num_jobs>
How many threads to use (can give good speed-ups with slow disks).

-l, --lang <lang>
Specify which languages to prepare, e.g., lhoste prepare librispeech mtedx_corpus data -l de -l fr -l es

Arguments

CORPUS_DIR
Required argument

OUTPUT_DIR
Required argument

musan

MUSAN data preparation.

```
lhotse prepare musan [OPTIONS] CORPUS_DIR OUTPUT_DIR
```

Options

--use-vocals, --no-vocals
Whether to include vocal music in "music" part.

Arguments

CORPUS_DIR
Required argument

OUTPUT_DIR
Required argument

nsc

This is a data preparation recipe for the National Corpus of Speech in Singaporean English.

```
lhotse prepare nsc [OPTIONS] CORPUS_DIR OUTPUT_DIR
```

Options

-p, --dataset-part <dataset_part>
Which part of NSC should be prepared

Options PART3_SameCloseMic | PART3_SeparateIVR

Arguments

CORPUS_DIR
Required argument

OUTPUT_DIR
Required argument

peoples-speech

Prepare The People's Speech corpus manifests.

```
lhotse prepare peoples-speech [OPTIONS] CORPUS_DIR OUTPUT_DIR
```

Arguments

CORPUS_DIR
Required argument

OUTPUT_DIR
Required argument

rir-noise

RIRS and noises data preparation.

```
lhotse prepare rir-noise [OPTIONS] CORPUS_DIR OUTPUT_DIR
```

Options

-p, --parts <parts>
Parts to prepare.

Default point_noise, iso_noise, real_rir, sim_rir

Arguments

CORPUS_DIR
Required argument

OUTPUT_DIR
Required argument

spgispeech

SPGISpeech ASR data preparation.

```
lhotse prepare spgispeech [OPTIONS] CORPUS_DIR OUTPUT_DIR
```

Options

-j, --num-jobs <num_jobs>
How many threads to use (can give good speed-ups with slow disks).

--normalize-text, --no-normalize-text
Normalize the text.

Arguments

CORPUS_DIR
Required argument

OUTPUT_DIR
Required argument

switchboard

The Switchboard corpus preparation.

This is conversational telephone speech collected as 2-channel, 8kHz-sampled data. We are using just the Switchboard-1 Phase 1 training data. The catalog number LDC97S62 (Switchboard-1 Release 2) corresponds, we believe, to what we have. We also use the Mississippi State transcriptions, which we download separately from http://www.isip.piconepress.com/projects/switchboard/releases/switchboard_word_alignments.tar.gz

This data is not available for free - your institution needs to have an LDC subscription.

```
lhotse prepare switchboard [OPTIONS] AUDIO_DIR OUTPUT_DIR
```

Options

--transcript-dir <transcript_dir>

--sentiment-dir <sentiment_dir>
Optional path to LDC2020T14 package with sentiment annotations for SWBD.

--omit-silence, --retain-silence
Should the [silence] segments be kept.

--absolute-paths <absolute_paths>
Whether to return absolute or relative (to the corpus dir) paths for recordings.

Arguments

AUDIO_DIR
Required argument

OUTPUT_DIR
Required argument

tedlium

TED-LIUM v3 recording and supervision manifest preparation.

```
lhotse prepare tedlium [OPTIONS] TEDLIUM_DIR OUTPUT_DIR
```

Arguments

TEDLIUM_DIR
Required argument

OUTPUT_DIR
Required argument

timit

TIMIT data preparation.

param corpus_dir Pathlike, the path of the data dir.

param output_dir Pathlike, the path where to write and save the manifests.

```
lhotse prepare timit [OPTIONS] CORPUS_DIR OUTPUT_DIR
```

Options

-p, --num-phones <num_phones>
The number of phones (60, 48 or 39) for modeling. And 48 is regarded as the default value.

-j, --num-jobs <num_jobs>
How many threads to use (can give good speed-ups with slow disks).

Arguments

CORPUS_DIR
Required argument

OUTPUT_DIR
Required argument

vctk

VCTK data preparation.

```
lhotse prepare vctk [OPTIONS] CORPUS_DIR OUTPUT_DIR
```

Arguments

CORPUS_DIR
Required argument

OUTPUT_DIR
Required argument

voxceleb

The VoxCeleb corpus preparation.

VoxCeleb is an audio-visual dataset consisting of short clips of human speech, extracted from interview videos uploaded to YouTube. VoxCeleb contains speech from speakers spanning a wide range of different ethnicities, accents, professions and ages. There are a total of 7000+ speakers and 1 million utterances.

```
lhotse prepare voxceleb [OPTIONS] OUTPUT_DIR
```

Options

-v1, --voxceleb1 <voxceleb1>
Path to VoxCeleb1 dataset.

-v2, --voxceleb2 <voxceleb2>
Path to VoxCeleb2 dataset.

-j, --num-jobs <num_jobs>
Number of parallel jobs.

Arguments

OUTPUT_DIR
Required argument

wenet-speech

The WenetSpeech corpus preparation.

```
lhotse prepare wenet-speech [OPTIONS] CORPUS_DIR OUTPUT_DIR
```

Options

-p, --dataset-parts <dataset_parts>
List of dataset parts to prepare. To prepare multiple parts,pass each with *-p* Example: *-p M -p TEST_NET*

-j, --num-jobs <num_jobs>
How many threads to use (can give good speed-ups with slow disks).

Arguments

CORPUS_DIR
Required argument

OUTPUT_DIR
Required argument

yesno

yes_no data preparation.

```
lhotse prepare yesno [OPTIONS] CORPUS_DIR OUTPUT_DIR
```

Arguments

CORPUS_DIR
Required argument

OUTPUT_DIR
Required argument

8.1.12 split

Load MANIFEST, split it into NUM_SPLITS equal parts and save as separate manifests in OUTPUT_DIR.

When your manifests are very large, prefer to use "lhotse split-lazy" instead.

```
lhotse split [OPTIONS] NUM_SPLITS MANIFEST OUTPUT_DIR
```

Options

-s, --shuffle
Optionally shuffle the sequence before splitting.

--pad, --no-pad
Whether to pad the split output idx with zeros (e.g. 01, 02, .., 10).

Arguments

NUM_SPLITS
Required argument

MANIFEST
Required argument

OUTPUT_DIR
Required argument

8.1.13 split-lazy

Load MANIFEST (lazily if in JSONL format) and split it into parts, each with CHUNK_SIZE items. The parts are saved to separate files with pattern "{output_dir}/{manifest.stem}.{chunk_idx}.jsonl.gz".

Prefer this to "lhotse split" when your manifests are very large.

```
lhotse split-lazy [OPTIONS] MANIFEST OUTPUT_DIR CHUNK_SIZE
```

Arguments

MANIFEST
Required argument

OUTPUT_DIR
Required argument

CHUNK_SIZE
Required argument

8.1.14 subset

Load MANIFEST, select the FIRST or LAST number of items and store it in OUTPUT_MANIFEST.

```
lhotse subset [OPTIONS] MANIFEST OUTPUT_MANIFEST
```

Options

--first <first>

--last <last>

--cutids <cutids>
A json string or path to json file containing array of cutids strings. E.g. –cutids '["cutid1", "cutid2"]'.

Arguments

MANIFEST
Required argument

OUTPUT_MANIFEST
Required argument

8.1.15 validate

Validate a Lhotse manifest file.

```
lhotse validate [OPTIONS] MANIFEST
```

Options

--read-data, --dont-read-data
Should the audio/features data be read from disk to perform additional checks (could be extremely slow for large manifests).

Arguments

MANIFEST
Required argument

8.1.16 validate-pair

Validate a pair of Lhotse RECORDINGS and SUPERVISIONS manifest files. Checks whether the two manifests are consistent with each other.

```
lhotse validate-pair [OPTIONS] RECORDINGS SUPERVISIONS
```

Options

--read-data, --dont-read-data
Should the audio/features data be read from disk to perform additional checks (could be extremely slow for large manifests).

Arguments

RECORDINGS
Required argument

SUPERVISIONS
Required argument

CHAPTER NINE

API REFERENCE

This page contains a comprehensive list of all classes and functions within *lhotse*.

9.1 Recording manifests

Data structures used for describing audio recordings in a dataset.

`lhotse.audio.`**`set_audio_duration_mismatch_tolerance`**`(`*delta*`)`

Override Lhotse's global threshold for allowed audio duration mismatch between the manifest and the actual data.

Some scenarios when a mismatch can happen:

- **the `Recording` manifest duration is rounded off too much** (i.e., bad user input, but too inconvenient to go back and fix the manifests)
- data augmentation changes the number of samples a bit in a difficult to predict way

When there is a mismatch, Lhotse will either trim or replicate the diff to match the value found in the `Recording` manifest.

Note: We don't recommend setting this lower than the default value, as it could break some data augmentation transforms.

Example:

```
>>> import lhotse
>>> lhotse.set_audio_duration_mismatch_tolerance(0.01)  # 10ms tolerance
```

Parameters **`delta`** (`float`) – New tolerance in seconds.

Return type `None`

`class` `lhotse.audio.`**`AudioSource`**`(`*type*, *channels*, *source*`)`

AudioSource represents audio data that can be retrieved from somewhere. Supported sources of audio are currently: - 'file' (formats supported by soundfile, possibly multi-channel) - 'command' [unix pipe] (must be WAVE, possibly multi-channel) - 'url' (any URL type that is supported by "smart_open" library, e.g. http/https/s3/gcp/azure/etc.) - 'memory' (any format, read from a binary string attached to 'source' member of AudioSource)

`type: str`

`channels: List[int]`

source: Union[str, bytes]

load_audio(*offset=0.0*, *duration=None*, *force_opus_sampling_rate=None*)

Load the AudioSource (from files, commands, or URLs) with soundfile, accounting for many audio formats and multi-channel inputs. Returns numpy array with shapes: (n_samples,) for single-channel, (n_channels, n_samples) for multi-channel.

Note: The elements in the returned array are in the range [-1.0, 1.0] and are of dtype *np.float32*.

Parameters **force_opus_sampling_rate** (`Optional[int]`) – This parameter is only used when we detect an OPUS file. It will tell ffmpeg to resample OPUS to this sampling rate.

Return type `ndarray`

with_path_prefix(*path*)

Return type *AudioSource*

to_dict()

Return type `dict`

static from_dict(*data*)

Return type *AudioSource*

__init__(*type*, *channels*, *source*)

class lhotse.audio.**Recording**(*id*, *sources*, *sampling_rate*, *num_samples*, *duration*, *transforms=None*)

The *Recording* manifest describes the recordings in a given corpus. It contains information about the recording, such as its path(s), duration, the number of samples, etc. It allows to represent multiple channels coming from one or more files.

This manifest does not specify any segmentation information or supervision such as the transcript or the speaker – we use *SupervisionSegment* for that.

Note that *Recording* can represent both a single utterance (e.g., in LibriSpeech) and a 1-hour session with multiple channels and speakers (e.g., in AMI). In the latter case, it is partitioned into data suitable for model training using *Cut*.

Hint: Lhotse reads audio recordings using **`pysoundfile`_** and **`audioread`_**, similarly to librosa, to support multiple audio formats. For OPUS files we require ffmpeg to be installed.

Hint: Since we support importing Kaldi data dirs, if `wav.scp` contains unix pipes, *Recording* will also handle them correctly.

Examples

A *Recording* can be simply created from a local audio file:

```
>>> from lhotse import RecordingSet, Recording, AudioSource
>>> recording = Recording.from_file('meeting.wav')
>>> recording
Recording(
    id='meeting',
```

(continues on next page)

(continued from previous page)

```
    sources=[AudioSource(type='file', channels=[0], source='meeting.wav')],
    sampling_rate=16000,
    num_samples=57600000,
    duration=3600.0,
    transforms=None
)
```

This manifest can be easily converted to a Python dict and serialized to JSON/JSONL/YAML/etc:

```
>>> recording.to_dict()
{'id': 'meeting',
 'sources': [{'type': 'file',
   'channels': [0],
   'source': 'meeting.wav'}],
 'sampling_rate': 16000,
 'num_samples': 57600000,
 'duration': 3600.0}
```

Recordings can be also created programatically, e.g. when they refer to URLs stored in S3 or somewhere else:

```
>>> s3_audio_files = ['s3://my-bucket/123-5678.flac', ...]
>>> recs = RecordingSet.from_recordings(
...     Recording(
...         id=url.split('/')[-1].replace('.flac', ''),
...         sources=[AudioSource(type='url', source=url, channels=[0])],
...         sampling_rate=16000,
...         num_samples=get_num_samples(url),
...         duration=get_duration(url)
...     )
...     for url in s3_audio_files
... )
```

It allows reading a subset of the audio samples as a numpy array:

```
>>> samples = recording.load_audio()
>>> assert samples.shape == (1, 16000)
>>> samples2 = recording.load_audio(offset=0.5)
>>> assert samples2.shape == (1, 8000)
```

id: str

sources: List[*lhotse.audio.AudioSource*]

sampling_rate: int

num_samples: int

duration: float

transforms: Optional[List[Dict]] = None

static from_file(*path*, *recording_id=None*, *relative_path_depth=None*, *force_opus_sampling_rate=None*, *force_read_audio=False*)

Read an audio file's header and create the corresponding `Recording`. Suitable to use when each physical file represents a separate recording session.

> **Caution:** If a recording session consists of multiple files (e.g. one per channel), it is advisable to create the `Recording` object manually, with each file represented as a separate `AudioSource` object.

Parameters

- **path** (`Union[Path, str]`) – Path to an audio file supported by libsoundfile (pysoundfile).
- **recording_id** (`Union[str, Callable[[Path], str], None]`) – recording id, when not specified ream the filename's stem ("x.wav" -> "x"). It can be specified as a string or a function that takes the recording path and returns a string.
- **relative_path_depth** (`Optional[int]`) – optional int specifying how many last parts of the file path should be retained in the `AudioSource`. By default writes the path as is.
- **force_opus_sampling_rate** (`Optional[int]`) – when specified, this value will be used as the sampling rate instead of the one we read from the manifest. This is useful for OPUS files that always have 48kHz rate and need to be resampled to the real one – we will perform that operation "under-the-hood". For non-OPUS files this input is undefined.
- **force_read_audio** (`bool`) – Set it to `True` for audio files that do not have any metadata in their headers (e.g., "The People's Speech" FLAC files).

Return type *Recording*

Returns a new `Recording` instance pointing to the audio file.

static from_bytes(*data*, *recording_id*)

Like *Recording.from_file()*, but creates a manifest for a byte string with raw encoded audio data. This data is first decoded to obtain info such as the sampling rate, number of channels, etc. Then, the binary data is attached to the manifest. Calling *Recording.load_audio()* does not perform any I/O and instead decodes the byte string contents in memory.

> **Note:** Intended use of this method is for packing Recordings into archives where metadata and data should be available together (e.g., in WebDataset style tarballs).

> **Caution:** Manifest created with this method cannot be stored as JSON because JSON doesn't allow serializing binary data.

Parameters

- **data** (`bytes`) – bytes, byte string containing encoded audio contents.
- **recording_id** (`str`) – recording id, unique string identifier.

Return type *Recording*

Returns a new `Recording` instance that owns the byte string data.

move_to_memory(*channels=None*, *offset=None*, *duration=None*, *format=None*)

Read audio data and return a copy of the manifest with binary data attached. Calling *Recording.load_audio()* on that copy will not trigger I/O.

If all arguments are left as defaults, we won't decode the audio and attach the bytes we read from disk/other source as-is. If `channels`, `duration`, or `offset` are specified, we'll decode the audio and re-encode it

into `format` before attaching. The default format is FLAC, other formats compatible with torchaudio.save are also accepted.

Return type *Recording*

to_dict()

Return type dict

to_cut()

Create a Cut out of this recording.

For single-channel recordings, we return a `MonoCut`. For multi-channel recordings, we return a `MixedCut` with as many tracks as the number of channels. The implementation of the multi-channel case may change in the future…

property num_channels

property channel_ids

load_audio(*channels=None*, *offset=0.0*, *duration=None*)

Read the audio samples from the underlying audio source (path, URL, unix pipe/command).

Parameters

- **channels** (Union[int, List[int], None]) – int or iterable of ints, a subset of channel IDs to read (reads all by default).
- **offset** (float) – seconds, where to start reading the audio (at offset 0 by default). Note that it is only efficient for local filesystem files, i.e. URLs and commands will read all the samples first and discard the unneeded ones afterwards.
- **duration** (Optional[float]) – seconds, indicates the total audio time to read (starting from `offset`).

Return type ndarray

Returns a numpy array of audio samples with shape `(num_channels, num_samples)`.

with_path_prefix(*path*)

Return type *Recording*

perturb_speed(*factor*, *affix_id=True*)

Return a new `Recording` that will lazily perturb the speed while loading audio. The `num_samples` and `duration` fields are updated to reflect the shrinking/extending effect of speed.

Parameters

- **factor** (float) – The speed will be adjusted this many times (e.g. factor=1.1 means 1.1x faster).
- **affix_id** (bool) – When true, we will modify the `Recording.id` field by affixing it with "_sp{factor}".

Return type *Recording*

Returns a modified copy of the current `Recording`.

perturb_tempo(*factor*, *affix_id=True*)

Return a new `Recording` that will lazily perturb the tempo while loading audio.

Compared to speed perturbation, tempo preserves pitch. The `num_samples` and `duration` fields are updated to reflect the shrinking/extending effect of tempo.

Parameters

- **factor** (float) – The tempo will be adjusted this many times (e.g. factor=1.1 means 1.1x faster).
- **affix_id** (bool) – When true, we will modify the Recording.id field by affixing it with "_tp{factor}".

Return type *Recording*

Returns a modified copy of the current Recording.

perturb_volume(*factor*, *affix_id=True*)
Return a new Recording that will lazily perturb the volume while loading audio.

Parameters

- **factor** (float) – The volume scale to be applied (e.g. factor=1.1 means 1.1x louder).
- **affix_id** (bool) – When true, we will modify the Recording.id field by affixing it with "_tp{factor}".

Return type *Recording*

Returns a modified copy of the current Recording.

reverb_rir(*rir_recording*, *normalize_output=True*, *early_only=False*, *affix_id=True*, *rir_channels=None*)
Return a new Recording that will lazily apply reverberation based on provided impulse response while loading audio.

Parameters

- **rir_recording** (*Recording*) – The impulse response to be used.
- **normalize_output** (bool) – When true, output will be normalized to have energy as input.
- **early_only** (bool) – When true, only the early reflections (first 50 ms) will be used.
- **affix_id** (bool) – When true, we will modify the Recording.id field by affixing it with "_rvb".
- **rir_channels** (Optional[List[int]]) – The channels of the impulse response to be used (in case of multi-channel impulse responses). By default, only the first channel is used.

Return type *Recording*

Returns the perturbed Recording.

resample(*sampling_rate*)
Return a new Recording that will be lazily resampled while loading audio. :type sampling_rate: int :param sampling_rate: The new sampling rate. :rtype: *Recording* :return: A resampled Recording.

static **from_dict**(*data*)

Return type *Recording*

__init__(*id*, *sources*, *sampling_rate*, *num_samples*, *duration*, *transforms=None*)

class lhotse.audio.**RecordingSet**(*recordings=None*)
RecordingSet represents a collection of recordings, indexed by recording IDs. It does not contain any annotation such as the transcript or the speaker identity – just the information needed to retrieve a recording such as its path, URL, number of channels, and some recording metadata (duration, number of samples).

It also supports (de)serialization to/from YAML/JSON/etc. and takes care of mapping between rich Python classes and YAML/JSON/etc. primitives during conversion.

When coming from Kaldi, think of it as `wav.scp` on steroids: `RecordingSet` also has the information from *reco2dur* and *reco2num_samples*, is able to represent multi-channel recordings and read a specified subset of channels, and support reading audio files directly, via a unix pipe, or downloading them on-the-fly from a URL (HTTPS/S3/Azure/GCP/etc.).

Examples:

`RecordingSet` can be created from an iterable of `Recording` objects:

```
>>> from lhotse import RecordingSet
>>> audio_paths = ['123-5678.wav', ...]
>>> recs = RecordingSet.from_recordings(Recording.from_file(p) for p in␣
↪audio_paths)
```

As well as from a directory, which will be scanned recursively for files with parallel processing:

```
>>> recs2 = RecordingSet.from_dir('/data/audio', pattern='*.flac', num_
↪jobs=4)
```

It behaves similarly to a `dict`:

```
>>> '123-5678' in recs
True
>>> recording = recs['123-5678']
>>> for recording in recs:
>>>    pass
>>> len(recs)
127
```

It also provides some utilities for I/O:

```
>>> recs.to_file('recordings.jsonl')
>>> recs.to_file('recordings.json.gz')  # auto-compression
>>> recs2 = RecordingSet.from_file('recordings.jsonl')
```

Manipulation:

```
>>> longer_than_5s = recs.filter(lambda r: r.duration > 5)
>>> first_100 = recs.subset(first=100)
>>> split_into_4 = recs.split(num_splits=4)
>>> shuffled = recs.shuffle()
```

And lazy data augmentation/transformation, that requires to adjust some information in the manifest (e.g., `num_samples` or `duration`). Note that in the following examples, the audio is untouched – the operations are stored in the manifest, and executed upon reading the audio:

```
>>> recs_sp = recs.perturb_speed(factor=1.1)
>>> recs_vp = recs.perturb_volume(factor=2.)
>>> recs_rvb = recs.reverb_rir(rir_recs)
>>> recs_24k = recs.resample(24000)
```

__init__(*recordings=None*)

property data: Union[Dict[str, *lhotse.audio.Recording*], Iterable[*lhotse.audio.Recording*]]
Alias property for `self.recordings`

Return type Union[Dict[str, *Recording*], Iterable[*Recording*]]

property ids: Iterable[str]

Return type Iterable[str]

static from_recordings(*recordings*)

Return type *RecordingSet*

static from_items(*recordings*)
Function to be implemented by every sub-class of this mixin. It's expected to create a sub-class instance out of an iterable of items that are held by the sub-class (e.g., `CutSet.from_items(iterable_of_cuts)`).

Return type *RecordingSet*

static from_dir(*path*, *pattern*, *num_jobs=1*, *force_opus_sampling_rate=None*, *recording_id=None*)
Recursively scan a directory `path` for audio files that match the given `pattern` and create a *RecordingSet* manifest for them. Suitable to use when each physical file represents a separate recording session.

> **Caution:** If a recording session consists of multiple files (e.g. one per channel), it is advisable to create each *Recording* object manually, with each file represented as a separate *AudioSource* object, and then a *RecordingSet* that contains all the recordings.

Parameters

- **path** (Union[Path, str]) – Path to a directory of audio of files (possibly with sub-directories).
- **pattern** (str) – A bash-like pattern specifying allowed filenames, e.g. `*.wav` or `session1-*.flac`.
- **num_jobs** (int) – The number of parallel workers for reading audio files to get their meta-data.
- **force_opus_sampling_rate** (Optional[int]) – when specified, this value will be used as the sampling rate instead of the one we read from the manifest. This is useful for OPUS files that always have 48kHz rate and need to be resampled to the real one – we will perform that operation "under-the-hood". For non-OPUS files this input does nothing.
- **recording_id** (Optional[Callable[[Path], str]]) – A function which takes the audio file path and returns the recording ID. If not specified, the filename will be used as the recording ID.

Returns a new `Recording` instance pointing to the audio file.

static from_dicts(*data*)

Return type *RecordingSet*

to_dicts()

Return type Iterable[dict]

split(*num_splits*, *shuffle=False*, *drop_last=False*)
Split the RecordingSet into num_splits pieces of equal size.

Parameters

- **num_splits** (int) – Requested number of splits.
- **shuffle** (bool) – Optionally shuffle the recordings order first.
- **drop_last** (bool) – determines how to handle splitting when len(seq) is not divisible by num_splits. When False (default), the splits might have unequal lengths. When True, it may discard the last element in some splits to ensure they are equally long.

Return type List[*RecordingSet*]

Returns A list of RecordingSet pieces.

split_lazy(*output_dir*, *chunk_size*, *prefix=''*)
Splits a manifest (either lazily or eagerly opened) into chunks, each with chunk_size items (except for the last one, typically).

In order to be memory efficient, this implementation saves each chunk to disk in a .jsonl.gz format as the input manifest is sampled.

Note: For lowest memory usage, use load_manifest_lazy to open the input manifest for this method.

Parameters

- **output_dir** (Union[Path, str]) – directory where the split manifests are saved. Each manifest is saved at: {output_dir}/{prefix}.{split_idx}.jsonl.gz
- **chunk_size** (int) – the number of items in each chunk.
- **prefix** (str) – the prefix of each manifest.

Return type List[*RecordingSet*]

Returns a list of lazily opened chunk manifests.

subset(*first=None*, *last=None*)
Return a new RecordingSet according to the selected subset criterion. Only a single argument to subset is supported at this time.

Parameters

- **first** (Optional[int]) – int, the number of first recordings to keep.
- **last** (Optional[int]) – int, the number of last recordings to keep.

Return type *RecordingSet*

Returns a new RecordingSet with the subset results.

load_audio(*recording_id*, *channels=None*, *offset_seconds=0.0*, *duration_seconds=None*)

Return type ndarray

with_path_prefix(*path*)

Return type *RecordingSet*

num_channels(*recording_id*)

Return type int

sampling_rate(*recording_id*)

Return type int

num_samples(*recording_id*)

Return type int

duration(*recording_id*)

Return type float

perturb_speed(*factor*, *affix_id=True*)

Return a new RecordingSet that will lazily perturb the speed while loading audio. The num_samples and duration fields are updated to reflect the shrinking/extending effect of speed.

Parameters

- **factor** (float) – The speed will be adjusted this many times (e.g. factor=1.1 means 1.1x faster).
- **affix_id** (bool) – When true, we will modify the Recording.id field by affixing it with "_sp{factor}".

Return type *RecordingSet*

Returns a RecordingSet containing the perturbed Recording objects.

perturb_tempo(*factor*, *affix_id=True*)

Return a new RecordingSet that will lazily perturb the tempo while loading audio. The num_samples and duration fields are updated to reflect the shrinking/extending effect of tempo.

Parameters

- **factor** (float) – The speed will be adjusted this many times (e.g. factor=1.1 means 1.1x faster).
- **affix_id** (bool) – When true, we will modify the Recording.id field by affixing it with "_sp{factor}".

Return type *RecordingSet*

Returns a RecordingSet containing the perturbed Recording objects.

perturb_volume(*factor*, *affix_id=True*)

Return a new RecordingSet that will lazily perturb the volume while loading audio.

Parameters

- **factor** (float) – The volume scale to be applied (e.g. factor=1.1 means 1.1x louder).
- **affix_id** (bool) – When true, we will modify the Recording.id field by affixing it with "_sp{factor}".

Return type *RecordingSet*

Returns a RecordingSet containing the perturbed Recording objects.

reverb_rir(*rir_recordings*, *normalize_output=True*, *early_only=False*, *affix_id=True*, *rir_channels=[0]*)
Return a new RecordingSet that will lazily apply reverberation based on provided impulse responses while loading audio.

Parameters

- **rir_recordings** (*RecordingSet*) – The impulse responses to be used.
- **normalize_output** (bool) – When true, output will be normalized to have energy as input.
- **early_only** (bool) – When true, only the early reflections (first 50 ms) will be used.
- **affix_id** (bool) – When true, we will modify the Recording.id field by affixing it with "_rvb".
- **rir_channels** (List[int]) – The channels to be used for the RIRs (if multi-channel). Uses first channel by default.

Return type *RecordingSet*

Returns a RecordingSet containing the perturbed Recording objects.

resample(*sampling_rate*)
Apply resampling to all recordings in the RecordingSet and return a new RecordingSet. :type sampling_rate: int :param sampling_rate: The new sampling rate. :rtype: *RecordingSet* :return: a new RecordingSet with lazily resampled Recording objects.

filter(*predicate*)
Return a new manifest containing only the items that satisfy predicate. If the manifest is lazy, the filtering will also be applied lazily.

Parameters predicate (Callable[[~T], bool]) – a function that takes a cut as an argument and returns bool.

Returns a filtered manifest.

classmethod from_file(*path*)

Return type Any

classmethod from_json(*path*)

Return type Any

classmethod from_jsonl(*path*)

Return type Any

classmethod from_jsonl_lazy(*path*)
Read a JSONL manifest in a lazy manner, which opens the file but does not read it immediately. It is only suitable for sequential reads and iteration.

> **Warning:** Opening the manifest in this way might cause some methods that rely on random access to fail.

Return type Any

classmethod from_yaml(*path*)

Return type Any

property is_lazy: bool

Indicates whether this manifest was opened in lazy (read-on-the-fly) mode or not.

Return type bool

map(*transform_fn*)

Apply *transform_fn* to each item in this manifest and return a new manifest. If the manifest is opened lazy, the transform is also applied lazily.

Parameters **transform_fn** (Callable[[~T], ~T]) – A callable (function) that accepts a single item instance and returns a new (or the same) instance of the same type. E.g. with CutSet, callable accepts `Cut` and returns also `Cut`.

Returns a new `CutSet` with transformed cuts.

classmethod mux(**manifests*, *stop_early=False*, *weights=None*, *seed=0*)

Merges multiple manifest iterables into a new iterable by lazily multiplexing them during iteration time. If one of the iterables is exhausted before the others, we will keep iterating until all iterables are exhausted. This behavior can be changed with `stop_early` parameter.

Parameters

- **manifests** – iterables to be multiplexed. They can be either lazy or eager, but the resulting manifest will always be lazy.
- **stop_early** (bool) – should we stop the iteration as soon as we exhaust one of the manifests.
- **weights** (Optional[List[Union[int, float]]]) – an optional weight for each iterable, affects the probability of it being sampled. The weights are uniform by default. If lengths are known, it makes sense to pass them here for uniform distribution of items in the expectation.
- **seed** (int) – the random seed, ensures deterministic order across multiple iterations.

classmethod open_writer(*path*, *overwrite=True*)

Open a sequential writer that allows to store the manifests one by one, without the necessity of storing the whole manifest set in-memory. Supports writing to JSONL format (`.jsonl`), with optional gzip compression (`.jsonl.gz`).

Note: when `path` is `None`, we will return a `InMemoryWriter` instead has the same API but stores the manifests in memory. It is convenient when you want to make disk saving optional.

Example:

```
>>> from lhotse import RecordingSet
... recordings = [...]
... with RecordingSet.open_writer('recordings.jsonl.gz') as writer:
...     for recording in recordings:
...         writer.write(recording)
```

This writer can be useful for continuing to write files that were previously stopped – it will open the existing file and scan it for item IDs to skip writing them later. It can also be queried for existing IDs so that the user code may skip preparing the corresponding manifests.

Example:

```
>>> from lhotse import RecordingSet, Recording
... with RecordingSet.open_writer('recordings.jsonl.gz', overwrite=False) as␣
↪writer:
...     for path in Path('.').rglob('*.wav'):
...         recording_id = path.stem
...         if writer.contains(recording_id):
...             # Item already written previously - skip processing.
...             continue
...         # Item doesn't exist yet - run extra work to prepare the manifest
...         # and store it.
...         recording = Recording.from_file(path, recording_id=recording_id)
...         writer.write(recording)
```

Return type Union[SequentialJsonlWriter, InMemoryWriter]

repeat(*times=None*, *preserve_id=False*)

Return a new, lazily evaluated manifest that iterates over the original elements times number of times.

Parameters

- **times** (Optional[int]) – how many times to repeat (infinite by default).
- **preserve_id** (bool) – when True, we won't update the element ID with repeat number.

Returns a repeated manifest.

shuffle(*rng=None*, *buffer_size=10000*)

Shuffles the elements and returns a shuffled variant of self. If the manifest is opened lazily, performs shuffling on-the-fly with a fixed buffer size.

Parameters **rng** (Optional[Random]) – an optional instance of random.Random for precise control of randomness.

Returns a shuffled copy of self, or a manifest that is shuffled lazily.

to_eager()

Evaluates all lazy operations on this manifest, if any, and returns a copy that keeps all items in memory. If the manifest was "eager" already, this is a no-op and won't copy anything.

to_file(*path*)

Return type None

to_json(*path*)

Return type None

to_jsonl(*path*)

Return type None

to_yaml(*path*)

Return type None

class lhotse.audio.**AudioMixer**(*base_audio*, *sampling_rate*, *reference_energy=None*)

Utility class to mix multiple waveforms into a single one. It should be instantiated separately for each mixing session (i.e. each `MixedCut` will create a separate `AudioMixer` to mix its tracks). It is initialized with a numpy array of audio samples (typically float32 in [-1, 1] range) that represents the "reference" signal for the mix. Other signals can be mixed to it with different time offsets and SNRs using the `add_to_mix` method. The time offset is relative to the start of the reference signal (only positive values are supported). The SNR is relative to the energy of the signal used to initialize the `AudioMixer`.

__init__(*base_audio*, *sampling_rate*, *reference_energy=None*)

AudioMixer's constructor.

Parameters

- **base_audio** (`ndarray`) – A numpy array with the audio samples for the base signal (all the other signals will be mixed to it).
- **sampling_rate** (`int`) – Sampling rate of the audio.
- **reference_energy** (`Optional[float]`) – Optionally pass a reference energy value to compute SNRs against. This might be required when `base_audio` corresponds to zero-padding.

property **num_samples_total: int**

Return type `int`

property **unmixed_audio: List[numpy.ndarray]**

Return a list of numpy arrays with the shape (1, num_samples), where each track is zero padded and scaled adequately to the offsets and SNR used in `add_to_mix` call.

Return type `List[ndarray]`

property **mixed_audio: numpy.ndarray**

Return a numpy ndarray with the shape (1, num_samples) - a mono mix of the tracks supplied with `add_to_mix` calls.

Return type `ndarray`

add_to_mix(*audio*, *snr=None*, *offset=0.0*)

Add audio (only support mono-channel) of a new track into the mix. :type audio: `ndarray` :param audio: An array of audio samples to be mixed in. :type snr: `Optional[float]` :param snr: Signal-to-noise ratio, assuming *audio* represents noise (positive SNR - lower *audio* energy, negative SNR - higher *audio* energy) :type offset: `float` :param offset: How many seconds to shift *audio* in time. For mixing, the signal will be padded before the start with low energy values. :return:

lhotse.audio.**audio_energy**(*audio*)

Return type `float`

lhotse.audio.**read_audio**(*path_or_fd*, *offset=0.0*, *duration=None*, *force_opus_sampling_rate=None*)

Return type `Tuple[ndarray, int]`

class lhotse.audio.**LibsndfileCompatibleAudioInfo**(*channels*, *frames*, *samplerate*, *duration*)

property **channels**

Alias for field number 0

property **frames**

Alias for field number 1

property samplerate
Alias for field number 2

property duration
Alias for field number 3

count(*value*, /)
Return number of occurrences of value.

index(*value*, *start=0*, *stop=9223372036854775807*, /)
Return first index of value.

Raises ValueError if the value is not present.

lhotse.audio.**info**(*path*, *force_opus_sampling_rate=None*, *force_read_audio=False*)

Return type *LibsndfileCompatibleAudioInfo*

lhotse.audio.**torchaudio_info**(*path*)
Return an audio info data structure that's a compatible subset of `pysoundfile.info()` that we need to create a `Recording` manifest.

Return type *LibsndfileCompatibleAudioInfo*

lhotse.audio.**torchaudio_load**(*path_or_fd*, *offset=0*, *duration=None*)

Return type `Tuple[ndarray, int]`

lhotse.audio.**soundfile_load**(*path_or_fd*, *offset=0*, *duration=None*)

Return type `Tuple[ndarray, int]`

lhotse.audio.**audioread_info**(*path*)
Return an audio info data structure that's a compatible subset of `pysoundfile.info()` that we need to create a `Recording` manifest.

Return type *LibsndfileCompatibleAudioInfo*

lhotse.audio.**audioread_load**(*path_or_file*, *offset=0.0*, *duration=None*, *dtype=<class 'numpy.float32'>*)
Load an audio buffer using audioread. This loads one block at a time, and then concatenates the results.

This function is based on librosa: https://github.com/librosa/librosa/blob/main/librosa/core/audio.py#L180

lhotse.audio.**assert_and_maybe_fix_num_samples**(*audio*, *offset*, *duration*, *recording*)

Return type `ndarray`

lhotse.audio.**opus_info**(*path*, *force_opus_sampling_rate=None*)

Return type *LibsndfileCompatibleAudioInfo*

lhotse.audio.**read_opus**(*path*, *offset=0.0*, *duration=None*, *force_opus_sampling_rate=None*)
Reads OPUS files either using torchaudio or ffmpeg. Torchaudio is faster, but if unavailable for some reason, we fallback to a slower ffmpeg-based implementation.

Return type `Tuple[ndarray, int]`

Returns a tuple of audio samples and the sampling rate.

`lhotse.audio.`**`read_opus_torchaudio`**`(`*path*, *offset=0.0*, *duration=None*, *force_opus_sampling_rate=None*`)`

Reads OPUS files using torchaudio. This is just running `tochaudio.load()`, but we take care of extra resampling if needed.

Return type `Tuple[ndarray, int]`

Returns a tuple of audio samples and the sampling rate.

`lhotse.audio.`**`read_opus_ffmpeg`**`(`*path*, *offset=0.0*, *duration=None*, *force_opus_sampling_rate=None*`)`

Reads OPUS files using ffmpeg in a shell subprocess. Unlike audioread, correctly supports offsets and durations for reading short chunks. Optionally, we can force ffmpeg to resample to the true sampling rate (if we know it up-front).

Return type `Tuple[ndarray, int]`

Returns a tuple of audio samples and the sampling rate.

`lhotse.audio.`**`parse_channel_from_ffmpeg_output`**`(`*ffmpeg_stderr*`)`

Return type `str`

`lhotse.audio.`**`sph_info`**`(`*path*`)`

Return type *LibsndfileCompatibleAudioInfo*

`lhotse.audio.`**`read_sph`**`(`*sph_path*, *offset=0.0*, *duration=None*`)`

Reads SPH files using sph2pipe in a shell subprocess. Unlike audioread, correctly supports offsets and durations for reading short chunks.

Return type `Tuple[ndarray, int]`

Returns a tuple of audio samples and the sampling rate.

exception `lhotse.audio.`**`AudioLoadingError`**

`__init__``(`**args*, ***kwargs*`)`

`args`

`with_traceback``()`

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception `lhotse.audio.`**`DurationMismatchError`**

`__init__``(`**args*, ***kwargs*`)`

`args`

`with_traceback``()`

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

`lhotse.audio.`**`suppress_audio_loading_errors`**`(`*enabled=True*`)`

Context manager that suppresses errors related to audio loading. Emits warning to the console.

`lhotse.audio.`**`null_result_on_audio_loading_error`**`(`*func*`)`

This is a decorator that makes a function return None when reading audio with Lhotse failed.

Example:

```
>>> @null_result_on_audio_loading_error
... def func_loading_audio(rec):
...     audio = rec.load_audio()  # if this fails, will return None instead
...     return other_func(audio)
```

Another example:

```
>>> # crashes on loading audio
>>> audio = load_audio(cut)
>>> # does not crash on loading audio, return None instead
>>> maybe_audio: Optional = null_result_on_audio_loading_error(load_audio)(cut)
```

Return type `Callable`

9.2 Supervision manifests

Data structures used for describing supervisions in a dataset.

class `lhotse.supervision.`**`AlignmentItem`**(*symbol*, *start*, *duration*)

This class contains an alignment item, for example a word, along with its start time (w.r.t. the start of recording) and duration. It can potentially be used to store other kinds of alignment items, such as subwords, pdfid's etc.

We use dataclasses instead of namedtuples (even though they are potentially slower) because of a serialization bug in nested namedtuples and dataclasses in Python 3.7 (see this: https://alexdelorenzo.dev/programming/2018/08/09/bug-in-dataclass.html). We can revert to namedtuples if we bump up the Python requirement to 3.8+.

`symbol: str`

`start: float`

`duration: float`

`property end: float`

Return type `float`

`with_offset`(*offset*)

Return an identical `AlignmentItem`, but with the `offset` added to the `start` field.

Return type *`AlignmentItem`*

`perturb_speed`(*factor*, *sampling_rate*)

Return an `AlignmentItem` that has time boundaries matching the recording/cut perturbed with the same factor. See *`SupervisionSegment.perturb_speed()`* for details.

Return type *`AlignmentItem`*

`trim`(*end*, *start=0*)

See *`SupervisionSegment.trim()`*.

Return type *`AlignmentItem`*

`transform`(*transform_fn*)

Perform specified transformation on the alignment content.

Return type *`AlignmentItem`*

`__init__`(*symbol*, *start*, *duration*)

class lhotse.supervision.**SupervisionSegment**(*id*, *recording_id*, *start*, *duration*, *channel=0*, *text=None*, *language=None*, *speaker=None*, *gender=None*, *custom=None*, *alignment=None*)

SupervisionSegment represents a time interval (segment) annotated with some supervision labels and/or metadata, such as the transcription, the speaker identity, the language, etc.

Each supervision has unique id and always refers to a specific recording (via recording_id) and a specific channel (by default, 0). It's also characterized by the start time (relative to the beginning of a *Recording* or a *Cut*) and a duration, both expressed in seconds.

The remaining fields are all optional, and their availability depends on specific corpora. Since it is difficult to predict all possible types of metadata, the custom field (a dict) can be used to insert types of supervisions that are not supported out of the box.

SupervisionSegment may contain multiple types of alignments. The alignment field is a dict, indexed by alignment's type (e.g., word or phone), and contains a list of *AlignmentItem* objects – simple structures that contain a given symbol and its time interval. Alignments can be read from CTM files or created programatically.

Examples

A simple segment with no supervision information:

```
>>> from lhotse import SupervisionSegment
>>> sup0 = SupervisionSegment(
...     id='rec00001-sup00000', recording_id='rec00001',
...     start=0.5, duration=5.0, channel=0
... )
```

Typical supervision containing transcript, speaker ID, gender, and language:

```
>>> sup1 = SupervisionSegment(
...     id='rec00001-sup00001', recording_id='rec00001',
...     start=5.5, duration=3.0, channel=0,
...     text='transcript of the second segment',
...     speaker='Norman Dyhrentfurth', language='English', gender='M'
... )
```

Two supervisions denoting overlapping speech on two separate channels in a microphone array/multiple headsets (pay attention to start, duration, and channel):

```
>>> sup2 = SupervisionSegment(
...     id='rec00001-sup00002', recording_id='rec00001',
...     start=15.0, duration=5.0, channel=0,
...     text="i have incredibly good news for you",
...     speaker='Norman Dyhrentfurth', language='English', gender='M'
... )
>>> sup3 = SupervisionSegment(
...     id='rec00001-sup00003', recording_id='rec00001',
...     start=18.0, duration=3.0, channel=1,
...     text="say what",
...     speaker='Hervey Arman', language='English', gender='M'
... )
```

A supervision with a phone alignment:

```
>>> from lhotse.supervision import AlignmentItem
>>> sup4 = SupervisionSegment(
```

(continues on next page)

(continued from previous page)

```
...     id='rec00001-sup00004', recording_id='rec00001',
...     start=33.0, duration=1.0, channel=0,
...     text="ice",
...     speaker='Maryla Zechariah', language='English', gender='F'
...     alignment={
...         'phone': [
...             AlignmentItem(symbol='AY0', start=33.0, duration=0.6),
...             AlignmentItem(symbol='S', start=33.6, duration=0.4)
...         ]
...     }
... )
```

Converting SupervisionSegment to a dict:

```
>>> sup0.to_dict()
{'id': 'rec00001-sup00000', 'recording_id': 'rec00001', 'start': 0.5,
↪'duration': 5.0, 'channel': 0}
```

id: str

recording_id: str

start: float

duration: float

channel: int = 0

text: Optional[str] = None

language: Optional[str] = None

speaker: Optional[str] = None

gender: Optional[str] = None

custom: Optional[Dict[str, Any]] = None

alignment: Optional[Dict[str, List[*lhotse.supervision.AlignmentItem*]]] = None

property end: float

Return type float

with_offset(*offset*)

Return an identical SupervisionSegment, but with the offset added to the start field.

Return type *SupervisionSegment*

perturb_speed(*factor*, *sampling_rate*, *affix_id=True*)

Return a SupervisionSegment that has time boundaries matching the recording/cut perturbed with the same factor.

Parameters

- **factor** (float) – The speed will be adjusted this many times (e.g. factor=1.1 means 1.1x faster).
- **sampling_rate** (int) – The sampling rate is necessary to accurately perturb the start and duration (going through the sample counts).

- **affix_id** (bool) – When true, we will modify the id and recording_id fields by affixing it with "_sp{factor}".

Return type *SupervisionSegment*

Returns a modified copy of the current SupervisionSegment.

perturb_tempo(*factor*, *sampling_rate*, *affix_id=True*)
Return a SupervisionSegment that has time boundaries matching the recording/cut perturbed with the same factor.

Parameters

- **factor** (float) – The tempo will be adjusted this many times (e.g. factor=1.1 means 1.1x faster).
- **sampling_rate** (int) – The sampling rate is necessary to accurately perturb the start and duration (going through the sample counts).
- **affix_id** (bool) – When true, we will modify the id and recording_id fields by affixing it with "_tp{factor}".

Return type *SupervisionSegment*

Returns a modified copy of the current SupervisionSegment.

perturb_volume(*factor*, *affix_id=True*)
Return a SupervisionSegment with modified ids.

Parameters

- **factor** (float) – The volume will be adjusted this many times (e.g. factor=1.1 means 1.1x louder).
- **affix_id** (bool) – When true, we will modify the id and recording_id fields by affixing it with "_vp{factor}".

Return type *SupervisionSegment*

Returns a modified copy of the current SupervisionSegment.

reverb_rir(*affix_id=True*)
Return a SupervisionSegment with modified ids.

Parameters **affix_id** (bool) – When true, we will modify the id and recording_id fields by affixing it with "_rvb".

Return type *SupervisionSegment*

Returns a modified copy of the current SupervisionSegment.

trim(*end*, *start=0*)
Return an identical SupervisionSegment, but ensure that self.start is not negative (in which case it's set to 0) and self.end does not exceed the end parameter. If a *start* is optionally provided, the supervision is trimmed from the left (note that start should be relative to the cut times).

This method is useful for ensuring that the supervision does not exceed a cut's bounds, in which case pass cut.duration as the end argument, since supervision times are relative to the cut.

Return type *SupervisionSegment*

map(*transform_fn*)
Return a copy of the current segment, transformed with transform_fn.

Parameters **transform_fn** (Callable[[*SupervisionSegment*], *SupervisionSegment*]) – a function that takes a segment as input, transforms it and returns a new segment.

Return type *SupervisionSegment*

Returns a modified SupervisionSegment.

transform_text(*transform_fn*)

Return a copy of the current segment with transformed text field. Useful for text normalization, phonetic transcription, etc.

Parameters **transform_fn** (Callable[[str], str]) – a function that accepts a string and returns a string.

Return type *SupervisionSegment*

Returns a SupervisionSegment with adjusted text.

transform_alignment(*transform_fn*, *type='word'*)

Return a copy of the current segment with transformed alignment field. Useful for text normalization, phonetic transcription, etc.

Parameters

- **type** (Optional[str]) – alignment type to transform (key for alignment dict).
- **transform_fn** (Callable[[str], str]) – a function that accepts a string and returns a string.

Return type *SupervisionSegment*

Returns a SupervisionSegment with adjusted alignments.

to_dict()

Return type dict

static **from_dict**(*data*)

Return type *SupervisionSegment*

__init__(*id*, *recording_id*, *start*, *duration*, *channel=0*, *text=None*, *language=None*, *speaker=None*, *gender=None*, *custom=None*, *alignment=None*)

class lhotse.supervision.**SupervisionSet**(*segments*)

SupervisionSet represents a collection of segments containing some supervision information (see *SupervisionSegment*), that are indexed by segment IDs.

It acts as a Python dict, extended with an efficient find operation that indexes and caches the supervision segments in an interval tree. It allows to quickly find supervision segments that correspond to a specific time interval.

When coming from Kaldi, think of *SupervisionSet* as a segments file on steroids, that may also contain *text*, *utt2spk*, *utt2gender*, *utt2dur*, etc.

Examples

Building a *SupervisionSet*:

```
>>> from lhotse import SupervisionSet, SupervisionSegment
>>> sups = SupervisionSet.from_segments([SupervisionSegment(...), ...])
```

Writing/reading a *SupervisionSet*:

```
>>> sups.to_file('supervisions.jsonl.gz')
>>> sups2 = SupervisionSet.from_file('supervisions.jsonl.gz')
```

Using *SupervisionSet* like a dict:

```
>>> 'rec00001-sup00000' in sups
True
>>> sups['rec00001-sup00000']
SupervisionSegment(id='rec00001-sup00000', recording_id='rec00001', start=0.
↪5, ...)
>>> for segment in sups:
...     pass
```

Searching by `recording_id` and time interval:

```
>>> matched_segments = sups.find(recording_id='rec00001', start_after=17.0,␣
↪end_before=25.0)
```

Manipulation:

```
>>> longer_than_5s = sups.filter(lambda s: s.duration > 5)
>>> first_100 = sups.subset(first=100)
>>> split_into_4 = sups.split(num_splits=4)
>>> shuffled = sups.shuffle()
```

__init__(*segments*)

property data: Union[Dict[str, *lhotse.supervision.SupervisionSegment*], Iterable[*lhotse.supervision.SupervisionSegment*]]
Alias property for `self.segments`

Return type Union[Dict[str, *SupervisionSegment*], Iterable[*SupervisionSegment*]]

property ids: Iterable[str]

Return type Iterable[str]

static from_segments(*segments*)

Return type *SupervisionSet*

static from_items(*segments*)
Function to be implemented by every sub-class of this mixin. It's expected to create a sub-class instance out of an iterable of items that are held by the sub-class (e.g., `CutSet.from_items(iterable_of_cuts)`).

Return type *SupervisionSet*

static from_dicts(*data*)

Return type *SupervisionSet*

static from_rttm(*path*)
Read an RTTM file located at `path` (or an iterator) and create a *SupervisionSet* manifest for them. Can be used to create supervisions from custom RTTM files (see, for example, `lhotse.dataset.DiarizationDataset`).

```
>>> from lhotse import SupervisionSet
>>> sup1 = SupervisionSet.from_rttm('/path/to/rttm_file')
>>> sup2 = SupervisionSet.from_rttm(Path('/path/to/rttm_dir').rglob('ref_*'))
```

The following description is taken from the [dscore](https://github.com/nryant/dscore#rttm) toolkit:

Rich Transcription Time Marked (RTTM) files are space-delimited text files containing one turn per line, each line containing ten fields:

- `Type` – segment type; should always by `SPEAKER`
- `File ID` – file name; basename of the recording minus extension (e.g.,

`rec1_a`) - `Channel ID` – channel (1-indexed) that turn is on; should always be `1` - `Turn Onset` – onset of turn in seconds from beginning of recording - `Turn Duration` – duration of turn in seconds - `Orthography Field` – should always by `<NA>` - `Speaker Type` – should always be `<NA>` - `Speaker Name` – name of speaker of turn; should be unique within scope of each file - `Confidence Score` – system confidence (probability) that information is correct; should always be `<NA>` - `Signal Lookahead Time` – should always be `<NA>`

For instance:

SPEAKER CMU_20020319-1400_d01_NONE 1 130.430000 2.350 <NA> <NA> juliet <NA> <NA> SPEAKER CMU_20020319-1400_d01_NONE 1 157.610000 3.060 <NA> <NA> tbc <NA> <NA> SPEAKER CMU_20020319-1400_d01_NONE 1 130.490000 0.450 <NA> <NA> chek <NA> <NA>

Parameters **path** (`Union[Path, str, Iterable[Union[Path, str]]]`) – Path to RTTM file or an iterator of paths to RTTM files.

Return type *SupervisionSet*

Returns a new `SupervisionSet` instance containing segments from the RTTM file.

with_alignment_from_ctm(*ctm_file*, *type='word'*, *match_channel=False*)

Add alignments from CTM file to the supervision set.

Parameters

- **ctm** – Path to CTM file.
- **type** (`str`) – Alignment type (optional, default = *word*).
- **match_channel** (`bool`) – if True, also match channel between CTM and SupervisionSegment

Return type *SupervisionSet*

Returns A new SupervisionSet with AlignmentItem objects added to the segments.

write_alignment_to_ctm(*ctm_file*, *type='word'*)

Write alignments to CTM file.

Parameters

- **ctm_file** (`Union[Path, str]`) – Path to output CTM file (will be created if not exists)
- **type** (`str`) – Alignment type to write (default = *word*)

Return type None

to_dicts()

Return type Iterable[dict]

split(*num_splits*, *shuffle=False*, *drop_last=False*)
Split the SupervisionSet into num_splits pieces of equal size.

Parameters

- **num_splits** (int) – Requested number of splits.
- **shuffle** (bool) – Optionally shuffle the recordings order first.
- **drop_last** (bool) – determines how to handle splitting when len(seq) is not divisible by num_splits. When False (default), the splits might have unequal lengths. When True, it may discard the last element in some splits to ensure they are equally long.

Return type List[*SupervisionSet*]

Returns A list of SupervisionSet pieces.

split_lazy(*output_dir*, *chunk_size*, *prefix=''*)
Splits a manifest (either lazily or eagerly opened) into chunks, each with chunk_size items (except for the last one, typically).

In order to be memory efficient, this implementation saves each chunk to disk in a .jsonl.gz format as the input manifest is sampled.

Note: For lowest memory usage, use load_manifest_lazy to open the input manifest for this method.

Parameters

- **it** – any iterable of Lhotse manifests.
- **output_dir** (Union[Path, str]) – directory where the split manifests are saved. Each manifest is saved at: {output_dir}/{prefix}.{split_idx}.jsonl.gz
- **chunk_size** (int) – the number of items in each chunk.
- **prefix** (str) – the prefix of each manifest.

Return type List[*SupervisionSet*]

Returns a list of lazily opened chunk manifests.

subset(*first=None*, *last=None*)
Return a new SupervisionSet according to the selected subset criterion. Only a single argument to subset is supported at this time.

Parameters

- **first** (Optional[int]) – int, the number of first supervisions to keep.
- **last** (Optional[int]) – int, the number of last supervisions to keep.

Return type *SupervisionSet*

Returns a new SupervisionSet with the subset results.

transform_text(*transform_fn*)
Return a copy of the current SupervisionSet with the segments having a transformed text field. Useful for text normalization, phonetic transcription, etc.

Parameters **transform_fn** (Callable[[str], str]) – a function that accepts a string and returns a string.

Return type *SupervisionSet*

Returns a SupervisionSet with adjusted text.

transform_alignment(*transform_fn*, *type='word'*)

Return a copy of the current SupervisionSet with the segments having a transformed alignment field. Useful for text normalization, phonetic transcription, etc.

Parameters

- **transform_fn** (Callable[[str], str]) – a function that accepts a string and returns a string.
- **type** (str) – alignment type to transform (key for alignment dict).

Return type *SupervisionSet*

Returns a SupervisionSet with adjusted text.

find(*recording_id*, *channel=None*, *start_after=0*, *end_before=None*, *adjust_offset=False*, *tolerance=0.001*)

Return an iterable of segments that match the provided recording_id.

Parameters

- **recording_id** (str) – Desired recording ID.
- **channel** (Optional[int]) – When specified, return supervisions in that channel - otherwise, in all channels.
- **start_after** (float) – When specified, return segments that start after the given value.
- **end_before** (Optional[float]) – When specified, return segments that end before the given value.
- **adjust_offset** (bool) – When true, return segments as if the recordings had started at start_after. This is useful for creating Cuts. From a user perspective, when dealing with a Cut, it is no longer helpful to know when the supervisions starts in a recording - instead, it's useful to know when the supervision starts relative to the start of the Cut. In the anticipated use-case, start_after and end_before would be the beginning and end of a cut; this option converts the times to be relative to the start of the cut.
- **tolerance** (float) – Additional margin to account for floating point rounding errors when comparing segment boundaries.

Return type Iterable[*SupervisionSegment*]

Returns An iterator over supervision segments satisfying all criteria.

filter(*predicate*)

Return a new manifest containing only the items that satisfy predicate. If the manifest is lazy, the filtering will also be applied lazily.

Parameters **predicate** (Callable[[~T], bool]) – a function that takes a cut as an argument and returns bool.

Returns a filtered manifest.

classmethod from_file(*path*)

Return type Any

classmethod from_json(*path*)

Return type Any

classmethod from_jsonl(*path*)

Return type Any

classmethod from_jsonl_lazy(*path*)

Read a JSONL manifest in a lazy manner, which opens the file but does not read it immediately. It is only suitable for sequential reads and iteration.

> **Warning:** Opening the manifest in this way might cause some methods that rely on random access to fail.

Return type Any

classmethod from_yaml(*path*)

Return type Any

property is_lazy: bool

Indicates whether this manifest was opened in lazy (read-on-the-fly) mode or not.

Return type bool

map(*transform_fn*)

Apply *transform_fn* to each item in this manifest and return a new manifest. If the manifest is opened lazy, the transform is also applied lazily.

Parameters **transform_fn** (Callable[[~T], ~T]) – A callable (function) that accepts a single item instance and returns a new (or the same) instance of the same type. E.g. with CutSet, callable accepts Cut and returns also Cut.

Returns a new CutSet with transformed cuts.

classmethod mux(**manifests*, *stop_early=False*, *weights=None*, *seed=0*)

Merges multiple manifest iterables into a new iterable by lazily multiplexing them during iteration time. If one of the iterables is exhausted before the others, we will keep iterating until all iterables are exhausted. This behavior can be changed with stop_early parameter.

Parameters

- **manifests** – iterables to be multiplexed. They can be either lazy or eager, but the resulting manifest will always be lazy.
- **stop_early** (bool) – should we stop the iteration as soon as we exhaust one of the manifests.
- **weights** (Optional[List[Union[int, float]]]) – an optional weight for each iterable, affects the probability of it being sampled. The weights are uniform by default. If lengths are known, it makes sense to pass them here for uniform distribution of items in the expectation.
- **seed** (int) – the random seed, ensures deterministic order across multiple iterations.

classmethod open_writer(*path*, *overwrite=True*)

Open a sequential writer that allows to store the manifests one by one, without the necessity of storing the whole manifest set in-memory. Supports writing to JSONL format (.jsonl), with optional gzip compression (.jsonl.gz).

Note: when `path` is `None`, we will return a `InMemoryWriter` instead has the same API but stores the manifests in memory. It is convenient when you want to make disk saving optional.

Example:

```
>>> from lhotse import RecordingSet
... recordings = [...]
... with RecordingSet.open_writer('recordings.jsonl.gz') as writer:
...     for recording in recordings:
...         writer.write(recording)
```

This writer can be useful for continuing to write files that were previously stopped – it will open the existing file and scan it for item IDs to skip writing them later. It can also be queried for existing IDs so that the user code may skip preparing the corresponding manifests.

Example:

```
>>> from lhotse import RecordingSet, Recording
... with RecordingSet.open_writer('recordings.jsonl.gz', overwrite=False) as␣
↪writer:
...     for path in Path('.').rglob('*.wav'):
...         recording_id = path.stem
...         if writer.contains(recording_id):
...             # Item already written previously - skip processing.
...             continue
...         # Item doesn't exist yet - run extra work to prepare the manifest
...         # and store it.
...         recording = Recording.from_file(path, recording_id=recording_id)
...         writer.write(recording)
```

Return type `Union[SequentialJsonlWriter, InMemoryWriter]`

repeat(*times=None*, *preserve_id=False*)

Return a new, lazily evaluated manifest that iterates over the original elements `times` number of times.

Parameters

- **times** (`Optional[int]`) – how many times to repeat (infinite by default).
- **preserve_id** (`bool`) – when `True`, we won't update the element ID with repeat number.

Returns a repeated manifest.

shuffle(*rng=None*, *buffer_size=10000*)

Shuffles the elements and returns a shuffled variant of self. If the manifest is opened lazily, performs shuffling on-the-fly with a fixed buffer size.

Parameters **rng** (`Optional[Random]`) – an optional instance of `random.Random` for precise control of randomness.

Returns a shuffled copy of self, or a manifest that is shuffled lazily.

to_eager()

Evaluates all lazy operations on this manifest, if any, and returns a copy that keeps all items in memory. If the manifest was "eager" already, this is a no-op and won't copy anything.

to_file(*path*)

Return type None

to_json(*path*)

Return type None

to_jsonl(*path*)

Return type None

to_yaml(*path*)

Return type None

9.3 Feature extraction and manifests

Data structures and tools used for feature extraction and description.

9.3.1 Features API - extractor and manifests

class lhotse.features.base.**FeatureExtractor**(*config=None*)

The base class for all feature extractors in Lhotse. It is initialized with a config object, specific to a particular feature extraction method. The config is expected to be a dataclass so that it can be easily serialized.

All derived feature extractors must implement at least the following:

- a `name` class attribute (how are these features called, e.g. 'mfcc')
- a `config_type` class attribute that points to the configuration dataclass type
- the `extract` method,
- the `frame_shift` property.

Feature extractors that support feature-domain mixing should additionally specify two static methods:

- `compute_energy`, and
- `mix`.

By itself, the `FeatureExtractor` offers the following high-level methods that are not intended for overriding:

- `extract_from_samples_and_store`
- `extract_from_recording_and_store`

These methods run a larger feature extraction pipeline that involves data augmentation and disk storage.

name = None

config_type = None

__init__(*config=None*)

abstract extract(*samples*, *sampling_rate*)

Defines how to extract features using a numpy ndarray of audio samples and the sampling rate.

Return type ndarray

Returns a numpy ndarray representing the feature matrix.

abstract property frame_shift: float

Return type float

abstract feature_dim(*sampling_rate*)

Return type int

property device: Union[str, torch.device]

Return type Union[str, device]

static mix(*features_a*, *features_b*, *energy_scaling_factor_b*)

Perform feature-domain mix of two signals, a and b, and return the mixed signal.

Parameters

- **features_a** (ndarray) – Left-hand side (reference) signal.
- **features_b** (ndarray) – Right-hand side (mixed-in) signal.
- **energy_scaling_factor_b** (float) – A scaling factor for features_b energy. It is used to achieve a specific SNR. E.g. to mix with an SNR of 10dB when both features_a and features_b energies are 100, the features_b signal energy needs to be scaled by 0.1. Since different features (e.g. spectrogram, fbank, MFCC) require different combination of transformations (e.g. exp, log, sqrt, pow) to allow mixing of two signals, the exact place where to apply energy_scaling_factor_b to the signal is determined by the implementer.

Return type ndarray

Returns A mixed feature matrix.

static compute_energy(*features*)

Compute the total energy of a feature matrix. How the energy is computed depends on a particular type of features. It is expected that when implemented, compute_energy will never return zero.

Parameters **features** (ndarray) – A feature matrix.

Return type float

Returns A positive float value of the signal energy.

extract_batch(*samples*, *sampling_rate*)

Performs batch extraction. It is not guaranteed to be faster than *FeatureExtractor.extract()* – it depends on whether the implementation of a particular feature extractor supports accelerated batch computation.

Note: Unless overridden by child classes, it defaults to sequentially calling *FeatureExtractor.extract()* on the inputs.

Note: This method *should* support variable length inputs.

Return type Union[ndarray, Tensor, List[ndarray], List[Tensor]]

`extract_from_samples_and_store`(*samples*, *storage*, *sampling_rate*, *offset=0*, *channel=None*, *augment_fn=None*)

Extract the features from an array of audio samples in a full pipeline:

- optional audio augmentation;
- extract the features;
- save them to disk in a specified directory;
- return a `Features` object with a description of the extracted features.

Note, unlike in `extract_from_recording_and_store`, the returned `Features` object might not be suitable to store in a `FeatureSet`, as it does not reference any particular `Recording`. Instead, this method is useful when extracting features from cuts - especially `MixedCut` instances, which may be created from multiple recordings and channels.

Parameters

- **samples** (`ndarray`) – a numpy ndarray with the audio samples.
- **sampling_rate** (`int`) – integer sampling rate of `samples`.
- **storage** (*FeaturesWriter*) – a `FeaturesWriter` object that will handle storing the feature matrices.
- **offset** (`float`) – an offset in seconds for where to start reading the recording - when used for `Cut` feature extraction, must be equal to `Cut.start`.
- **channel** (`Optional[int]`) – an optional channel number to insert into `Features` manifest.
- **augment_fn** (`Optional[Callable[[ndarray, int], ndarray]]`) – an optional `WavAugmenter` instance to modify the waveform before feature extraction.

Return type *Features*

Returns a `Features` manifest item for the extracted feature matrix (it is not written to disk).

`extract_from_recording_and_store`(*recording*, *storage*, *offset=0*, *duration=None*, *channels=None*, *augment_fn=None*)

Extract the features from a `Recording` in a full pipeline:

- load audio from disk;
- optionally, perform audio augmentation;
- extract the features;
- save them to disk in a specified directory;
- return a `Features` object with a description of the extracted features and the source data used.

Parameters

- **recording** (*Recording*) – a `Recording` that specifies what's the input audio.
- **storage** (*FeaturesWriter*) – a `FeaturesWriter` object that will handle storing the feature matrices.
- **offset** (`float`) – an optional offset in seconds for where to start reading the recording.
- **duration** (`Optional[float]`) – an optional duration specifying how much audio to load from the recording.

- **channels** (Union[int, List[int], None]) – an optional int or list of ints, specifying the channels; by default, all channels will be used.
- **augment_fn** (Optional[Callable[[ndarray, int], ndarray]]) – an optional WavAugmenter instance to modify the waveform before feature extraction.

Return type *Features*

Returns a Features manifest item for the extracted feature matrix.

classmethod from_dict(*data*)

Return type *FeatureExtractor*

to_dict()

Return type Dict[str, Any]

classmethod from_yaml(*path*)

Return type *FeatureExtractor*

to_yaml(*path*)

lhotse.features.base.**get_extractor_type**(*name*)

Return the feature extractor type corresponding to the given name.

Parameters **name** (str) – specifies which feature extractor should be used.

Return type Type

Returns A feature extractors type.

lhotse.features.base.**create_default_feature_extractor**(*name*)

Create a feature extractor object with a default configuration.

Parameters **name** (str) – specifies which feature extractor should be used.

Return type Optional[*FeatureExtractor*]

Returns A new feature extractor instance.

lhotse.features.base.**register_extractor**(*cls*)

This decorator is used to register feature extractor classes in Lhotse so they can be easily created just by knowing their name.

An example of usage:

@register_extractor class MyFeatureExtractor: …

Parameters **cls** – A type (class) that is being registered.

Returns Registered type.

class lhotse.features.base.**TorchaudioFeatureExtractor**(*config=None*)

Common abstract base class for all torchaudio based feature extractors.

extract(*samples*, *sampling_rate*)

Defines how to extract features using a numpy ndarray of audio samples and the sampling rate.

Return type ndarray

Returns a numpy ndarray representing the feature matrix.

property frame_shift: float

Return type float

__init__(*config=None*)

static compute_energy(*features*)

Compute the total energy of a feature matrix. How the energy is computed depends on a particular type of features. It is expected that when implemented, `compute_energy` will never return zero.

Parameters **features** (ndarray) – A feature matrix.

Return type float

Returns A positive float value of the signal energy.

config_type = None

property device: Union[str, torch.device]

Return type Union[str, device]

extract_batch(*samples*, *sampling_rate*)

Performs batch extraction. It is not guaranteed to be faster than *FeatureExtractor.extract()* – it depends on whether the implementation of a particular feature extractor supports accelerated batch computation.

Note: Unless overridden by child classes, it defaults to sequentially calling *FeatureExtractor.extract()* on the inputs.

Note: This method *should* support variable length inputs.

Return type Union[ndarray, Tensor, List[ndarray], List[Tensor]]

extract_from_recording_and_store(*recording*, *storage*, *offset=0*, *duration=None*, *channels=None*, *augment_fn=None*)

Extract the features from a `Recording` in a full pipeline:

- load audio from disk;
- optionally, perform audio augmentation;
- extract the features;
- save them to disk in a specified directory;
- return a `Features` object with a description of the extracted features and the source data used.

Parameters

- **recording** (*Recording*) – a `Recording` that specifies what's the input audio.
- **storage** (*FeaturesWriter*) – a `FeaturesWriter` object that will handle storing the feature matrices.
- **offset** (float) – an optional offset in seconds for where to start reading the recording.
- **duration** (Optional[float]) – an optional duration specifying how much audio to load from the recording.

- **channels** (Union[int, List[int], None]) – an optional int or list of ints, specifying the channels; by default, all channels will be used.
- **augment_fn** (Optional[Callable[[ndarray, int], ndarray]]) – an optional WavAugmenter instance to modify the waveform before feature extraction.

Return type *Features*

Returns a Features manifest item for the extracted feature matrix.

extract_from_samples_and_store(*samples, storage, sampling_rate, offset=0, channel=None, augment_fn=None*)

Extract the features from an array of audio samples in a full pipeline:

- optional audio augmentation;
- extract the features;
- save them to disk in a specified directory;
- return a Features object with a description of the extracted features.

Note, unlike in extract_from_recording_and_store, the returned Features object might not be suitable to store in a FeatureSet, as it does not reference any particular Recording. Instead, this method is useful when extracting features from cuts - especially MixedCut instances, which may be created from multiple recordings and channels.

Parameters

- **samples** (ndarray) – a numpy ndarray with the audio samples.
- **sampling_rate** (int) – integer sampling rate of samples.
- **storage** (*FeaturesWriter*) – a FeaturesWriter object that will handle storing the feature matrices.
- **offset** (float) – an offset in seconds for where to start reading the recording - when used for Cut feature extraction, must be equal to Cut.start.
- **channel** (Optional[int]) – an optional channel number to insert into Features manifest.
- **augment_fn** (Optional[Callable[[ndarray, int], ndarray]]) – an optional WavAugmenter instance to modify the waveform before feature extraction.

Return type *Features*

Returns a Features manifest item for the extracted feature matrix (it is not written to disk).

abstract feature_dim(*sampling_rate*)

Return type int

classmethod from_dict(*data*)

Return type *FeatureExtractor*

classmethod from_yaml(*path*)

Return type *FeatureExtractor*

static mix(*features_a, features_b, energy_scaling_factor_b*)

Perform feature-domain mix of two signals, a and b, and return the mixed signal.

Parameters

- **features_a** (ndarray) – Left-hand side (reference) signal.
- **features_b** (ndarray) – Right-hand side (mixed-in) signal.
- **energy_scaling_factor_b** (float) – A scaling factor for features_b energy. It is used to achieve a specific SNR. E.g. to mix with an SNR of 10dB when both features_a and features_b energies are 100, the features_b signal energy needs to be scaled by 0.1. Since different features (e.g. spectrogram, fbank, MFCC) require different combination of transformations (e.g. exp, log, sqrt, pow) to allow mixing of two signals, the exact place where to apply energy_scaling_factor_b to the signal is determined by the implementer.

Return type ndarray

Returns A mixed feature matrix.

name = None

to_dict()

Return type Dict[str, Any]

to_yaml(*path*)

class lhotse.features.base.**Features**(*type*, *num_frames*, *num_features*, *frame_shift*, *sampling_rate*, *start*, *duration*, *storage_type*, *storage_path*, *storage_key*, *recording_id=None*, *channels=None*)

Represents features extracted for some particular time range in a given recording and channel. It contains metadata about how it's stored: storage_type describes "how to read it", for now it supports numpy arrays serialized with np.save, as well as arrays compressed with lilcom; storage_path is the path to the file on the local filesystem.

type: str

num_frames: int

num_features: int

frame_shift: float

sampling_rate: int

start: float

duration: float

storage_type: str

storage_path: str

storage_key: Union[str, bytes]

recording_id: Optional[str] = None

channels: Optional[Union[int, List[int]]] = None

property end: float

Return type float

load(*start=None*, *duration=None*)

Return type ndarray

move_to_memory(*start=0*, *duration=None*)

Return type *Features*

with_path_prefix(*path*)

Return type *Features*

to_dict()

Return type dict

copy_feats(*writer*)

Read the referenced feature array and save it using `writer`. Returns a copy of the manifest with updated fields related to the feature storage.

Return type *Features*

static **from_dict**(*data*)

Return type *Features*

__init__(*type*, *num_frames*, *num_features*, *frame_shift*, *sampling_rate*, *start*, *duration*, *storage_type*, *storage_path*, *storage_key*, *recording_id=None*, *channels=None*)

class lhotse.features.base.**FeatureSet**(*features=None*)

Represents a feature manifest, and allows to read features for given recordings within particular channels and time ranges. It also keeps information about the feature extractor parameters used to obtain this set. When a given recording/time-range/channel is unavailable, raises a KeyError.

__init__(*features=None*)

property **data**: Union[Dict[str, *lhotse.features.base.Features*], Iterable[*lhotse.features.base.Features*]]

Alias property for `self.features`

Return type Union[Dict[str, *Features*], Iterable[*Features*]]

to_eager()

Evaluates all lazy operations on this manifest, if any, and returns a copy that keeps all items in memory. If the manifest was "eager" already, this is a no-op and won't copy anything.

Return type *FeatureSet*

static **from_features**(*features*)

Return type *FeatureSet*

static **from_items**(*features*)

Function to be implemented by every sub-class of this mixin. It's expected to create a sub-class instance out of an iterable of items that are held by the sub-class (e.g., `CutSet.from_items(iterable_of_cuts)`).

Return type *FeatureSet*

static **from_dicts**(*data*)

Return type *FeatureSet*

`to_dicts()`

Return type Iterable[dict]

`with_path_prefix`(*path*)

Return type *FeatureSet*

`split`(*num_splits*, *shuffle=False*, *drop_last=False*)

Split the FeatureSet into num_splits pieces of equal size.

Parameters

- **num_splits** (int) – Requested number of splits.
- **shuffle** (bool) – Optionally shuffle the recordings order first.
- **drop_last** (bool) – determines how to handle splitting when len(seq) is not divisible by num_splits. When False (default), the splits might have unequal lengths. When True, it may discard the last element in some splits to ensure they are equally long.

Return type List[*FeatureSet*]

Returns A list of FeatureSet pieces.

`split_lazy`(*output_dir*, *chunk_size*, *prefix=''*)

Splits a manifest (either lazily or eagerly opened) into chunks, each with chunk_size items (except for the last one, typically).

In order to be memory efficient, this implementation saves each chunk to disk in a .jsonl.gz format as the input manifest is sampled.

Note: For lowest memory usage, use load_manifest_lazy to open the input manifest for this method.

Parameters

- **it** – any iterable of Lhotse manifests.
- **output_dir** (Union[Path, str]) – directory where the split manifests are saved. Each manifest is saved at: {output_dir}/{prefix}.{split_idx}.jsonl.gz
- **chunk_size** (int) – the number of items in each chunk.
- **prefix** (str) – the prefix of each manifest.

Return type List[*FeatureSet*]

Returns a list of lazily opened chunk manifests.

`shuffle`(**args*, ***kwargs*)

Shuffles the elements and returns a shuffled variant of self. If the manifest is opened lazily, performs shuffling on-the-fly with a fixed buffer size.

Parameters **rng** – an optional instance of random.Random for precise control of randomness.

Returns a shuffled copy of self, or a manifest that is shuffled lazily.

`subset`(*first=None*, *last=None*)

Return a new FeatureSet according to the selected subset criterion. Only a single argument to subset is supported at this time.

Parameters

- **first** (Optional[int]) – int, the number of first supervisions to keep.
- **last** (Optional[int]) – int, the number of last supervisions to keep.

Return type *FeatureSet*

Returns a new FeatureSet with the subset results.

find(*recording_id*, *channel_id=0*, *start=0.0*, *duration=None*, *leeway=0.05*)
Find and return a Features object that best satisfies the search criteria. Raise a KeyError when no such object is available.

Parameters

- **recording_id** (str) – str, requested recording ID.
- **channel_id** (int) – int, requested channel.
- **start** (float) – float, requested start time in seconds for the feature chunk.
- **duration** (Optional[float]) – optional float, requested duration in seconds for the feature chunk. By default, return everything from the start.
- **leeway** (float) – float, controls how strictly we have to match the requested start and duration criteria. It is necessary to keep a small positive value here (default 0.05s), as there might be differences between the duration of recording/supervision segment, and the duration of features. The latter one is constrained to be a multiple of frame_shift, while the former can be arbitrary.

Return type *Features*

Returns a Features object satisfying the search criteria.

load(*recording_id*, *channel_id=0*, *start=0.0*, *duration=None*)
Find a Features object that best satisfies the search criteria and load the features as a numpy ndarray. Raise a KeyError when no such object is available.

Return type ndarray

copy_feats(*writer*)
For each manifest in this FeatureSet, read the referenced feature array and save it using writer. Returns a copy of the manifest with updated fields related to the feature storage.

Return type *FeatureSet*

compute_global_stats(*storage_path=None*)
Compute the global means and standard deviations for each feature bin in the manifest. It follows the implementation in scikit-learn: https://github.com/scikit-learn/scikit-learn/blob/0fb307bf39bbdacd6ed713c00724f8f871d60370/sklearn/utils/extmath.py#L715 which follows the paper: "Algorithms for computing the sample variance: analysis and recommendations", by Chan, Golub, and LeVeque.

Parameters **storage_path** (Union[Path, str, None]) – an optional path to a file where the stats will be stored with pickle.

Return a dict of ``{'norm_means'``{'norm_means' np.ndarray, 'norm_stds': np.ndarray}``
with the shape of the arrays equal to the number of feature bins in this manifest.

Return type Dict[str, ndarray]

filter(*predicate*)
Return a new manifest containing only the items that satisfy predicate. If the manifest is lazy, the filtering will also be applied lazily.

Parameters **predicate** (Callable[[~T], bool]) – a function that takes a cut as an argument and returns bool.

Returns a filtered manifest.

classmethod **from_file**(*path*)

Return type Any

classmethod **from_json**(*path*)

Return type Any

classmethod **from_jsonl**(*path*)

Return type Any

classmethod **from_jsonl_lazy**(*path*)

Read a JSONL manifest in a lazy manner, which opens the file but does not read it immediately. It is only suitable for sequential reads and iteration.

> **Warning:** Opening the manifest in this way might cause some methods that rely on random access to fail.

Return type Any

classmethod **from_yaml**(*path*)

Return type Any

property **is_lazy**: bool

Indicates whether this manifest was opened in lazy (read-on-the-fly) mode or not.

Return type bool

map(*transform_fn*)

Apply *transform_fn* to each item in this manifest and return a new manifest. If the manifest is opened lazy, the transform is also applied lazily.

Parameters **transform_fn** (Callable[[~T], ~T]) – A callable (function) that accepts a single item instance and returns a new (or the same) instance of the same type. E.g. with CutSet, callable accepts Cut and returns also Cut.

Returns a new CutSet with transformed cuts.

classmethod **mux**(**manifests*, *stop_early=False*, *weights=None*, *seed=0*)

Merges multiple manifest iterables into a new iterable by lazily multiplexing them during iteration time. If one of the iterables is exhausted before the others, we will keep iterating until all iterables are exhausted. This behavior can be changed with stop_early parameter.

Parameters

- **manifests** – iterables to be multiplexed. They can be either lazy or eager, but the resulting manifest will always be lazy.
- **stop_early** (bool) – should we stop the iteration as soon as we exhaust one of the manifests.

- **weights** (Optional[List[Union[int, float]]]) – an optional weight for each iterable, affects the probability of it being sampled. The weights are uniform by default. If lengths are known, it makes sense to pass them here for uniform distribution of items in the expectation.
- **seed** (int) – the random seed, ensures deterministic order across multiple iterations.

classmethod open_writer(*path*, *overwrite=True*)

Open a sequential writer that allows to store the manifests one by one, without the necessity of storing the whole manifest set in-memory. Supports writing to JSONL format (.jsonl), with optional gzip compression (.jsonl.gz).

Note: when path is None, we will return a InMemoryWriter instead has the same API but stores the manifests in memory. It is convenient when you want to make disk saving optional.

Example:

```
>>> from lhotse import RecordingSet
... recordings = [...]
... with RecordingSet.open_writer('recordings.jsonl.gz') as writer:
...     for recording in recordings:
...         writer.write(recording)
```

This writer can be useful for continuing to write files that were previously stopped – it will open the existing file and scan it for item IDs to skip writing them later. It can also be queried for existing IDs so that the user code may skip preparing the corresponding manifests.

Example:

```
>>> from lhotse import RecordingSet, Recording
... with RecordingSet.open_writer('recordings.jsonl.gz', overwrite=False) as␣
↪writer:
...     for path in Path('.').rglob('*.wav'):
...         recording_id = path.stem
...         if writer.contains(recording_id):
...             # Item already written previously - skip processing.
...             continue
...         # Item doesn't exist yet - run extra work to prepare the manifest
...         # and store it.
...         recording = Recording.from_file(path, recording_id=recording_id)
...         writer.write(recording)
```

Return type Union[SequentialJsonlWriter, InMemoryWriter]

repeat(*times=None*, *preserve_id=False*)

Return a new, lazily evaluated manifest that iterates over the original elements times number of times.

Parameters

- **times** (Optional[int]) – how many times to repeat (infinite by default).
- **preserve_id** (bool) – when True, we won't update the element ID with repeat number.

Returns a repeated manifest.

to_file(*path*)

Return type None

to_json(*path*)

Return type None

to_jsonl(*path*)

Return type None

to_yaml(*path*)

Return type None

class lhotse.features.base.**FeatureSetBuilder**(*feature_extractor*, *storage*, *augment_fn=None*)

An extended constructor for the FeatureSet. Think of it as a class wrapper for a feature extraction script. It consumes an iterable of Recordings, extracts the features specified by the FeatureExtractor config, and saves stores them on the disk.

Eventually, we plan to extend it with the capability to extract only the features in specified regions of recordings and to perform some time-domain data augmentation.

__init__(*feature_extractor*, *storage*, *augment_fn=None*)

process_and_store_recordings(*recordings*, *output_manifest=None*, *num_jobs=1*)

Return type *FeatureSet*

lhotse.features.base.**store_feature_array**(*feats*, *storage*)

Store `feats` array on disk, using `lilcom` compression by default.

Parameters

- **feats** (ndarray) – a numpy ndarray containing features.
- **storage** (*FeaturesWriter*) – a `FeaturesWriter` object to use for array storage.

Return type str

Returns a path to the file containing the stored array.

lhotse.features.base.**compute_global_stats**(*feature_manifests*, *storage_path=None*)

Compute the global means and standard deviations for each feature bin in the manifest. It performs only a single pass over the data and iteratively updates the estimate of the means and variances.

We follow the implementation in scikit-learn: https://github.com/scikit-learn/scikit-learn/blob/0fb307bf39bbdacd6ed713c00724f8f871d60370/sklearn/utils/extmath.py#L715 which follows the paper: "Algorithms for computing the sample variance: analysis and recommendations", by Chan, Golub, and LeVeque.

Parameters

- **feature_manifests** (Iterable[*Features*]) – an iterable of `Features` objects.
- **storage_path** (Union[Path, str, None]) – an optional path to a file where the stats will be stored with pickle.

Return a dict of ``{'norm_means'``{'norm_means' np.ndarray, 'norm_stds': np.ndarray}`` with the shape of the arrays equal to the number of feature bins in this manifest.

Return type Dict[str, ndarray]

9.3.2 Lhotse's feature extractors

class lhotse.features.kaldi.extractors.**Fbank**(*config=None*)

name = 'kaldi-fbank'

config_type
alias of lhotse.features.kaldi.extractors.FbankConfig

__init__(*config=None*)

property frame_shift: float

Return type float

feature_dim(*sampling_rate*)

Return type int

extract(*samples, sampling_rate*)
Defines how to extract features using a numpy ndarray of audio samples and the sampling rate.

Return type Union[ndarray, Tensor]

Returns a numpy ndarray representing the feature matrix.

static mix(*features_a, features_b, energy_scaling_factor_b*)
Perform feature-domain mix of two signals, a and b, and return the mixed signal.

Parameters

- **features_a** (ndarray) – Left-hand side (reference) signal.
- **features_b** (ndarray) – Right-hand side (mixed-in) signal.
- **energy_scaling_factor_b** (float) – A scaling factor for features_b energy. It is used to achieve a specific SNR. E.g. to mix with an SNR of 10dB when both features_a and features_b energies are 100, the features_b signal energy needs to be scaled by 0.1. Since different features (e.g. spectrogram, fbank, MFCC) require different combination of transformations (e.g. exp, log, sqrt, pow) to allow mixing of two signals, the exact place where to apply energy_scaling_factor_b to the signal is determined by the implementer.

Return type ndarray

Returns A mixed feature matrix.

static compute_energy(*features*)
Compute the total energy of a feature matrix. How the energy is computed depends on a particular type of features. It is expected that when implemented, compute_energy will never return zero.

Parameters **features** (ndarray) – A feature matrix.

Return type float

Returns A positive float value of the signal energy.

class lhotse.features.kaldi.extractors.**Mfcc**(*config=None*)

name = 'kaldi-mfcc'

config_type
alias of lhotse.features.kaldi.extractors.MfccConfig

__init__(*config=None*)

property frame_shift: float

Return type float

feature_dim(*sampling_rate*)

Return type int

extract(*samples, sampling_rate*)

Defines how to extract features using a numpy ndarray of audio samples and the sampling rate.

Return type Union[ndarray, Tensor]

Returns a numpy ndarray representing the feature matrix.

9.3.3 Kaldi feature extractors as network layers

This whole module is authored and contributed by Jesus Villalba, with minor changes by Piotr Żelasko to make it more consistent with Lhotse.

It contains a PyTorch implementation of feature extractors that is very close to Kaldi's – notably, it differs in that the preemphasis and DC offset removal are applied in the time, rather than frequency domain. This should not significantly affect any results, as confirmed by Jesus.

This implementation works well with autograd and batching, and can be used neural network layers.

Update January 2022: These modules now expose a new API function called "online_inference" that may be used to compute the features when the audio is streaming. The implementation is stateless, and passes the waveform remainders back to the user to feed them to the modules once new data becomes available. The implementation is compatible with JIT scripting via TorchScript.

class lhotse.features.kaldi.layers.**Wav2Win**(*sampling_rate=16000, frame_length=0.025, frame_shift=0.01, pad_length=None, remove_dc_offset=True, preemph_coeff=0.97, window_type='povey', dither=0.0, snip_edges=False, energy_floor=1e-10, raw_energy=True, return_log_energy=False*)

Apply standard Kaldi preprocessing (dithering, removing DC offset, pre-emphasis, etc.) on the input waveforms and partition them into overlapping frames (of audio samples). Note: no feature extraction happens in here, the output is still a time-domain signal.

Example:

```
>>> x = torch.randn(1, 16000, dtype=torch.float32)
>>> x.shape
torch.Size([1, 16000])
>>> t = Wav2Win()
>>> t(x).shape
torch.Size([1, 100, 400])
```

The input is a tensor of shape (batch_size, num_samples). The output is a tensor of shape (batch_size, num_frames, window_length). When return_log_energy==True, returns a tuple where the second element is a log-energy tensor of shape (batch_size, num_frames).

__init__(*sampling_rate=16000*, *frame_length=0.025*, *frame_shift=0.01*, *pad_length=None*, *remove_dc_offset=True*, *preemph_coeff=0.97*, *window_type='povey'*, *dither=0.0*, *snip_edges=False*, *energy_floor=1e-10*, *raw_energy=True*, *return_log_energy=False*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

Return type `Tuple`[`Tensor`, `Optional`[`Tensor`]]

online_inference(*x*, *context=None*)

The same as the `forward()` method, except it accepts an extra argument with the remainder waveform from the previous call of `online_inference()`, and returns a tuple of `((frames, log_energy), remainder)`.

Return type `Tuple`[`Tuple`[`Tensor`, `Optional`[`Tensor`]], `Tensor`]

T_destination

alias of TypeVar('T_destination', bound=`Mapping`[`str`, `torch.Tensor`])

add_module(*name*, *module*)

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Args:

name (string): name of the child module. The child module can be accessed from this module using the given name

module (Module): child module to be added to the module.

Return type `None`

apply(*fn*)

Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also nn-init-doc).

Args: fn (`Module` -> None): function to be applied to each submodule

Returns: Module: self

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
```

(continues on next page)

(continued from previous page)

```
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

Return type ~T

bfloat16()

Casts all floating point parameters and buffers to `bfloat16` datatype.

Note: This method modifies the module in-place.

Returns: Module: self

Return type ~T

buffers(*recurse=True*)

Returns an iterator over module buffers.

Args:

recurse (bool): if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields: torch.Tensor: module buffer

Example:

```
>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

Return type `Iterator[Tensor]`

children()

Returns an iterator over immediate children modules.

Yields: Module: a child module

Return type `Iterator[Module]`

`cpu()`

Moves all model parameters and buffers to the CPU.

Note: This method modifies the module in-place.

Returns: Module: self

Return type ~T

`cuda(device=None)`

Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Note: This method modifies the module in-place.

Args:

device (int, optional): if specified, all parameters will be copied to that device

Returns: Module: self

Return type ~T

`double()`

Casts all floating point parameters and buffers to `double` datatype.

Note: This method modifies the module in-place.

Returns: Module: self

Return type ~T

`dump_patches: bool = False`

This allows better BC support for *load_state_dict()*. In *state_dict()*, the version number will be saved as in the attribute *_metadata* of the returned state dict, and thus pickled. *_metadata* is a dictionary with keys that follow the naming convention of state dict. See `_load_from_state_dict` on how to use this information in loading.

If new parameters/buffers are added/removed from a module, this number shall be bumped, and the module's *_load_from_state_dict* method can compare the version number and do appropriate changes if the state dict is from before the change.

`eval()`

Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between *.eval()* and several similar mechanisms that may be confused with it.

Returns: Module: self

Return type ~T

`extra_repr()`

Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

Return type `str`

`float()`

Casts all floating point parameters and buffers to `float` datatype.

Note: This method modifies the module in-place.

Returns: Module: self

Return type ~T

`get_buffer(`*target*`)`

Returns the buffer given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Args:

target: The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns: torch.Tensor: The buffer referenced by `target`

Raises:

AttributeError: If the target string references an invalid path or resolves to something that is not a buffer

Return type `Tensor`

`get_extra_state()`

Returns any extra state to include in the module's state_dict. Implement this and a corresponding `set_extra_state()` for your module if you need to store extra state. This function is called when building the module's *state_dict()*.

Note that extra state should be pickleable to ensure working serialization of the state_dict. We only provide provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

Returns: object: Any extra state to store in the module's state_dict

Return type `Any`

`get_parameter`(*target*)

Returns the parameter given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Args:

target: The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns: torch.nn.Parameter: The Parameter referenced by `target`

Raises:

AttributeError: If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

Return type `Parameter`

`get_submodule`(*target*)

Returns the submodule given by `target` if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module A` that looks like this:

(The diagram shows an `nn.Module A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in `target`. A query against `named_modules` achieves the same result, but it is O(N) in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

Args:

target: The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

Returns: torch.nn.Module: The submodule referenced by `target`

Raises:

AttributeError: If the target string references an invalid path or resolves to something that is not an `nn.Module`

Return type `Module`

`half`()

Casts all floating point parameters and buffers to `half` datatype.

Note: This method modifies the module in-place.

Returns: Module: self

Return type ~T

load_state_dict(*state_dict*, *strict=True*)
Copies parameters and buffers from *state_dict* into this module and its descendants. If strict is True, then the keys of *state_dict* must exactly match the keys returned by this module's state_dict() function.

Args:

state_dict (dict): a dict containing parameters and persistent buffers.

strict (bool, optional): whether to strictly enforce that the keys in *state_dict* match the keys returned by this module's state_dict() function. Default: True

Returns:

NamedTuple with missing_keys and unexpected_keys fields:

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

Note: If a parameter or buffer is registered as None and its corresponding key exists in *state_dict*, *load_state_dict()* will raise a RuntimeError.

modules()
Returns an iterator over all modules in the network.

Yields: Module: a module in the network

Note: Duplicate modules are returned only once. In the following example, l will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
        print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

Return type Iterator[Module]

named_buffers(*prefix=''*, *recurse=True*)
Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Args: prefix (str): prefix to prepend to all buffer names. recurse (bool): if True, then yields buffers of this module

and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields: (string, torch.Tensor): Tuple containing the name and buffer

Example:

```
>>> for name, buf in self.named_buffers():
>>>    if name in ['running_var']:
>>>        print(buf.size())
```

Return type `Iterator[Tuple[str, Tensor]]`

named_children()

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields: (string, Module): Tuple containing a name and child module

Example:

```
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

Return type `Iterator[Tuple[str, Module]]`

named_modules(*memo=None*, *prefix=''*, *remove_duplicate=True*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Args: memo: a memo to store the set of modules already added to the result prefix: a prefix that will be added to the name of the module remove_duplicate: whether to remove the duplicated module instances in the result

or not

Yields: (string, Module): Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
        print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

named_parameters(*prefix=''*, *recurse=True*)

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Args: prefix (str): prefix to prepend to all parameter names. recurse (bool): if True, then yields parameters of this module

and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: (string, Parameter): Tuple containing the name and parameter

Example:

```
>>> for name, param in self.named_parameters():
>>>    if name in ['bias']:
>>>        print(param.size())
```

Return type `Iterator[Tuple[str, Parameter]]`

parameters(*recurse=True*)

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Args:

recurse (bool): if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: Parameter: module parameter

Example:

```
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

Return type `Iterator[Parameter]`

register_backward_hook(*hook*)

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `RemovableHandle`

register_buffer(*name, tensor, persistent=True*)

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's *`state_dict`*.

Buffers can be accessed as attributes using given names.

Args:

name (string): name of the buffer. The buffer can be accessed from this module using the given name

tensor (Tensor or None): buffer to be registered. If `None`, then operations that run on buffers, such as *`cuda`*, are ignored. If `None`, the buffer is **not** included in the module's *`state_dict`*.

persistent (bool): whether the buffer is part of this module's *`state_dict`*.

Example:

```
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

Return type None

register_forward_hook(*hook*)

Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the forward. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after *forward()* is called.

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling handle.remove()

Return type RemovableHandle

register_forward_pre_hook(*hook*)

Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the forward. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling handle.remove()

Return type RemovableHandle

register_full_backward_hook(*hook*)

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to module inputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The grad_input and grad_output are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of grad_input in subsequent computations. grad_input will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in grad_input and grad_output will be None for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module's forward function.

> **Warning:** Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

Returns:

`torch.utils.hooks.RemovableHandle`: a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `RemovableHandle`

`register_module`(*name*, *module*)

Alias for *`add_module()`*.

Return type `None`

`register_parameter`(*name*, *param*)

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Args:

name (string): name of the parameter. The parameter can be accessed from this module using the given name

param (Parameter or None): parameter to be added to the module. If `None`, then operations that run on parameters, such as *`cuda`*, are ignored. If `None`, the parameter is **not** included in the module's *`state_dict`*.

Return type `None`

`requires_grad_`(*requires_grad=True*)

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See locally-disable-grad-doc for a comparison between *.requires_grad_()* and several similar mechanisms that may be confused with it.

Args:

requires_grad (bool): whether autograd should record operations on parameters in this module. Default: `True`.

Returns: Module: self

Return type ~T

`set_extra_state`(*state*)

This function is called from *`load_state_dict()`* to handle any extra state found within the *state_dict*. Implement this function and a corresponding *`get_extra_state()`* for your module if you need to store extra state within its *state_dict*.

Args: state (dict): Extra state from the *state_dict*

`share_memory`()

See `torch.Tensor.share_memory_()`

Return type ~T

`state_dict`(*destination=None*, *prefix=''*, *keep_vars=False*)

Returns a dictionary containing a whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to `None` are not included.

Returns:

dict: a dictionary containing a whole state of the module

Example:

```
>>> module.state_dict().keys()
['bias', 'weight']
```

`to`(**args*, ***kwargs*)

Moves and/or casts the parameters and buffers.

This can be called as

`to`(*device=None*, *dtype=None*, *non_blocking=False*)

`to`(*dtype*, *non_blocking=False*)

`to`(*tensor*, *non_blocking=False*)

`to`(*memory_format=torch.channels_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex `dtypes`. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Args:

device (`torch.device`): the desired device of the parameters and buffers in this module

dtype (`torch.dtype`): the desired floating point or complex dtype of the parameters and buffers in this module

tensor (torch.Tensor): Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module

memory_format (`torch.memory_format`): the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns: Module: self

Examples:

```
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
```

(continues on next page)

(continued from previous page)

```
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[ 0.3741+0.j,  0.2382+0.j],
        [ 0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)
```

to_empty(*, *device*)

Moves the parameters and buffers to the specified device without copying storage.

Args:

device (torch.device): The desired device of the parameters and buffers in this module.

Returns: Module: self

Return type ~T

train(*mode=True*)

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

Args:

mode (bool): whether to set training mode (True) or evaluation mode (`False`). Default: `True`.

Returns: Module: self

Return type ~T

`type`(*dst_type*)
Casts all parameters and buffers to `dst_type`.

Note: This method modifies the module in-place.

Args: dst_type (type or string): the desired type

Returns: Module: self

Return type ~T

`xpu`(*device=None*)
Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

Note: This method modifies the module in-place.

Arguments:

device (int, optional): if specified, all parameters will be copied to that device

Returns: Module: self

Return type ~T

`zero_grad`(*set_to_none=False*)
Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

Args:

set_to_none (bool): instead of setting to zero, set the grads to None. See `torch.optim.Optimizer.zero_grad()` for details.

Return type `None`

`training: bool`

class `lhotse.features.kaldi.layers.Wav2FFT`(*sampling_rate=16000*, *frame_length=0.025*, *frame_shift=0.01*, *round_to_power_of_two=True*, *remove_dc_offset=True*, *preemph_coeff=0.97*, *window_type='povey'*, *dither=0.0*, *snip_edges=False*, *energy_floor=1e-10*, *raw_energy=True*, *use_energy=True*)
Apply standard Kaldi preprocessing (dithering, removing DC offset, pre-emphasis, etc.) on the input waveforms and compute their Short-Time Fourier Transform (STFT). The output is a complex-valued tensor.

Example:

```
>>> x = torch.randn(1, 16000, dtype=torch.float32)
>>> x.shape
torch.Size([1, 16000])
>>> t = Wav2FFT()
>>> t(x).shape
torch.Size([1, 100, 257])
```

The input is a tensor of shape `(batch_size, num_samples)`. The output is a tensor of shape `(batch_size, num_frames, num_fft_bins)` with dtype `torch.complex64`.

__init__(*sampling_rate=16000*, *frame_length=0.025*, *frame_shift=0.01*, *round_to_power_of_two=True*, *remove_dc_offset=True*, *preemph_coeff=0.97*, *window_type='povey'*, *dither=0.0*, *snip_edges=False*, *energy_floor=1e-10*, *raw_energy=True*, *use_energy=True*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

property sampling_rate: int

Return type `int`

property frame_length: float

Return type `float`

property frame_shift: float

Return type `float`

property remove_dc_offset: bool

Return type `bool`

property preemph_coeff: float

Return type `float`

property window_type: str

Return type `str`

property dither: float

Return type `float`

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

Return type `Tensor`

online_inference(*x*, *context=None*)

Return type `Tuple`[`Tensor`, `Tensor`]

T_destination

alias of TypeVar('T_destination', bound=`Mapping`[`str`, `torch.Tensor`])

add_module(*name*, *module*)

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Args:

name (string): name of the child module. The child module can be accessed from this module using the given name

module (Module): child module to be added to the module.

Return type `None`

apply(*fn*)

Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also nn-init-doc).

Args: fn (`Module` -> None): function to be applied to each submodule

Returns: Module: self

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

Return type ~T

bfloat16()

Casts all floating point parameters and buffers to `bfloat16` datatype.

Note: This method modifies the module in-place.

Returns: Module: self

Return type ~T

buffers(*recurse=True*)

Returns an iterator over module buffers.

Args:

recurse (bool): if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields: torch.Tensor: module buffer

Example:

```
>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

Return type `Iterator[Tensor]`

children()

Returns an iterator over immediate children modules.

Yields: Module: a child module

Return type `Iterator[Module]`

cpu()

Moves all model parameters and buffers to the CPU.

Note: This method modifies the module in-place.

Returns: Module: self

Return type ~T

cuda(*device=None*)

Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Note: This method modifies the module in-place.

Args:

device (int, optional): if specified, all parameters will be copied to that device

Returns: Module: self

Return type ~T

double()
Casts all floating point parameters and buffers to `double` datatype.

Note: This method modifies the module in-place.

Returns: Module: self

Return type ~T

dump_patches: bool = False
This allows better BC support for *load_state_dict()*. In *state_dict()*, the version number will be saved as in the attribute *_metadata* of the returned state dict, and thus pickled. *_metadata* is a dictionary with keys that follow the naming convention of state dict. See `_load_from_state_dict` on how to use this information in loading.

If new parameters/buffers are added/removed from a module, this number shall be bumped, and the module's *_load_from_state_dict* method can compare the version number and do appropriate changes if the state dict is from before the change.

eval()
Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between *.eval()* and several similar mechanisms that may be confused with it.

Returns: Module: self

Return type ~T

extra_repr()
Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

Return type `str`

float()
Casts all floating point parameters and buffers to `float` datatype.

Note: This method modifies the module in-place.

Returns: Module: self

Return type ~T

get_buffer(*target*)
Returns the buffer given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Args:

target: The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns: torch.Tensor: The buffer referenced by `target`

Raises:

AttributeError: If the target string references an invalid path or resolves to something that is not a buffer

Return type `Tensor`

`get_extra_state()`

Returns any extra state to include in the module's state_dict. Implement this and a corresponding `set_extra_state()` for your module if you need to store extra state. This function is called when building the module's *state_dict()*.

Note that extra state should be pickleable to ensure working serialization of the state_dict. We only provide provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

Returns: object: Any extra state to store in the module's state_dict

Return type `Any`

`get_parameter`(*target*)

Returns the parameter given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Args:

target: The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns: torch.nn.Parameter: The Parameter referenced by `target`

Raises:

AttributeError: If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

Return type `Parameter`

`get_submodule`(*target*)

Returns the submodule given by `target` if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module` `A` that looks like this:

(The diagram shows an `nn.Module` `A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in `target`. A query against `named_modules` achieves the same result, but it is O(N) in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

Args:

target: The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

Returns: torch.nn.Module: The submodule referenced by `target`

Raises:

AttributeError: If the target string references an invalid path or resolves to something that is not an `nn.Module`

Return type `Module`

half()

Casts all floating point parameters and buffers to `half` datatype.

Note: This method modifies the module in-place.

Returns: Module: self

Return type ~T

load_state_dict(*state_dict*, *strict=True*)

Copies parameters and buffers from *state_dict* into this module and its descendants. If `strict` is `True`, then the keys of *state_dict* must exactly match the keys returned by this module's `state_dict()` function.

Args:

state_dict (dict): a dict containing parameters and persistent buffers.

strict (bool, optional): whether to strictly enforce that the keys in *state_dict* match the keys returned by this module's `state_dict()` function. Default: `True`

Returns:

`NamedTuple` with `missing_keys` and `unexpected_keys` fields:

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

Note: If a parameter or buffer is registered as `None` and its corresponding key exists in *state_dict*, *load_state_dict()* will raise a `RuntimeError`.

modules()

Returns an iterator over all modules in the network.

Yields: Module: a module in the network

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
        print(idx, '->', m)
```

(continues on next page)

(continued from previous page)

```
0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

Return type `Iterator[Module]`

named_buffers(*prefix=''*, *recurse=True*)
Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Args: prefix (str): prefix to prepend to all buffer names. recurse (bool): if True, then yields buffers of this module

and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields: (string, torch.Tensor): Tuple containing the name and buffer

Example:

```
>>> for name, buf in self.named_buffers():
>>>    if name in ['running_var']:
>>>        print(buf.size())
```

Return type `Iterator[Tuple[str, Tensor]]`

named_children()
Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields: (string, Module): Tuple containing a name and child module

Example:

```
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

Return type `Iterator[Tuple[str, Module]]`

named_modules(*memo=None*, *prefix=''*, *remove_duplicate=True*)
Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Args: memo: a memo to store the set of modules already added to the result prefix: a prefix that will be added to the name of the module remove_duplicate: whether to remove the duplicated module instances in the result

or not

Yields: (string, Module): Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
        print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

named_parameters(*prefix=''*, *recurse=True*)

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Args: prefix (str): prefix to prepend to all parameter names. recurse (bool): if True, then yields parameters of this module

and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: (string, Parameter): Tuple containing the name and parameter

Example:

```
>>> for name, param in self.named_parameters():
>>>    if name in ['bias']:
>>>        print(param.size())
```

Return type Iterator[Tuple[str, Parameter]]

parameters(*recurse=True*)

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Args:

recurse (bool): if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: Parameter: module parameter

Example:

```
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

Return type Iterator[Parameter]

register_backward_hook(*hook*)

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `RemovableHandle`

register_buffer(*name*, *tensor*, *persistent=True*)
Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's *state_dict*.

Buffers can be accessed as attributes using given names.

Args:

name (string): name of the buffer. The buffer can be accessed from this module using the given name

tensor (Tensor or None): buffer to be registered. If None, then operations that run on buffers, such as *cuda*, are ignored. If `None`, the buffer is **not** included in the module's *state_dict*.

persistent (bool): whether the buffer is part of this module's *state_dict*.

Example:

```
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

Return type `None`

register_forward_hook(*hook*)
Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after *forward()* is called.

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `RemovableHandle`

register_forward_pre_hook(*hook*)
Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns:

`torch.utils.hooks.RemovableHandle`: a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `RemovableHandle`

`register_full_backward_hook`(*hook*)

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to module inputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The `grad_input` and `grad_output` are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in `grad_input` and `grad_output` will be `None` for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module's forward function.

> **Warning:** Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

Returns:

`torch.utils.hooks.RemovableHandle`: a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `RemovableHandle`

`register_module`(*name*, *module*)

Alias for `add_module()`.

Return type `None`

`register_parameter`(*name*, *param*)

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Args:

name (string): name of the parameter. The parameter can be accessed from this module using the given name

param (Parameter or None): parameter to be added to the module. If `None`, then operations that run on parameters, such as `cuda`, are ignored. If `None`, the parameter is **not** included in the module's `state_dict`.

Return type None

requires_grad_(*requires_grad=True*)

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See locally-disable-grad-doc for a comparison between *.requires_grad_()* and several similar mechanisms that may be confused with it.

Args:

requires_grad (bool): whether autograd should record operations on parameters in this module. Default: `True`.

Returns: Module: self

Return type ~T

set_extra_state(*state*)

This function is called from `load_state_dict()` to handle any extra state found within the *state_dict*. Implement this function and a corresponding `get_extra_state()` for your module if you need to store extra state within its *state_dict*.

Args: state (dict): Extra state from the *state_dict*

share_memory()

See `torch.Tensor.share_memory_()`

Return type ~T

state_dict(*destination=None*, *prefix=''*, *keep_vars=False*)

Returns a dictionary containing a whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to `None` are not included.

Returns:

dict: a dictionary containing a whole state of the module

Example:

```
>>> module.state_dict().keys()
['bias', 'weight']
```

to(**args*, ***kwargs*)

Moves and/or casts the parameters and buffers.

This can be called as

to(*device=None*, *dtype=None*, *non_blocking=False*)

to(*dtype*, *non_blocking=False*)

to(*tensor*, *non_blocking=False*)

to(*memory_format=torch.channels_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex `dtypes`. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Args:

device (`torch.device`): the desired device of the parameters and buffers in this module

dtype (`torch.dtype`): the desired floating point or complex dtype of the parameters and buffers in this module

tensor (torch.Tensor): Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module

memory_format (`torch.memory_format`): the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns: Module: self

Examples:

```
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
```

(continues on next page)

(continued from previous page)

```
Parameter containing:
tensor([[ 0.3741+0.j,  0.2382+0.j],
        [ 0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)
```

to_empty(*, *device*)

Moves the parameters and buffers to the specified device without copying storage.

Args:

device (torch.device): The desired device of the parameters and buffers in this module.

Returns: Module: self

Return type ~T

train(*mode=True*)

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

Args:

mode (bool): whether to set training mode (True) or evaluation mode (`False`). Default: `True`.

Returns: Module: self

Return type ~T

type(*dst_type*)

Casts all parameters and buffers to `dst_type`.

Note: This method modifies the module in-place.

Args: dst_type (type or string): the desired type

Returns: Module: self

Return type ~T

xpu(*device=None*)

Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

Note: This method modifies the module in-place.

Arguments:

device (int, optional): if specified, all parameters will be copied to that device

Returns: Module: self

Return type ~T

`zero_grad`(*set_to_none=False*)

Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

Args:

set_to_none (bool): instead of setting to zero, set the grads to None. See `torch.optim.Optimizer.zero_grad()` for details.

Return type `None`

`training: bool`

class `lhotse.features.kaldi.layers.Wav2Spec`(*sampling_rate=16000, frame_length=0.025, frame_shift=0.01, round_to_power_of_two=True, remove_dc_offset=True, preemph_coeff=0.97, window_type='povey', dither=0.0, snip_edges=False, energy_floor=1e-10, raw_energy=True, use_energy=True, use_fft_mag=False*)

Apply standard Kaldi preprocessing (dithering, removing DC offset, pre-emphasis, etc.) on the input waveforms and compute their Short-Time Fourier Transform (STFT). The STFT is transformed either to a magnitude spectrum (`use_fft_mag=True`) or a power spectrum (`use_fft_mag=False`).

Example:

```
>>> x = torch.randn(1, 16000, dtype=torch.float32)
>>> x.shape
torch.Size([1, 16000])
>>> t = Wav2Spec()
>>> t(x).shape
torch.Size([1, 100, 257])
```

The input is a tensor of shape `(batch_size, num_samples)`. The output is a tensor of shape `(batch_size, num_frames, num_fft_bins)`.

`__init__`(*sampling_rate=16000, frame_length=0.025, frame_shift=0.01, round_to_power_of_two=True, remove_dc_offset=True, preemph_coeff=0.97, window_type='povey', dither=0.0, snip_edges=False, energy_floor=1e-10, raw_energy=True, use_energy=True, use_fft_mag=False*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

`T_destination`

alias of TypeVar('T_destination', bound=`Mapping[str, torch.Tensor]`)

`add_module`(*name, module*)

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Args:

name (string): name of the child module. The child module can be accessed from this module using the given name

module (Module): child module to be added to the module.

Return type `None`

apply(*fn*)

Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also nn-init-doc).

Args: fn (`Module` -> None): function to be applied to each submodule

Returns: Module: self

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

Return type ~T

bfloat16()

Casts all floating point parameters and buffers to `bfloat16` datatype.

Note: This method modifies the module in-place.

Returns: Module: self

Return type ~T

buffers(*recurse=True*)

Returns an iterator over module buffers.

Args:

recurse (bool): if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields: torch.Tensor: module buffer

Example:

```
>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

Return type `Iterator[Tensor]`

children()
Returns an iterator over immediate children modules.

Yields: Module: a child module

Return type `Iterator[Module]`

cpu()
Moves all model parameters and buffers to the CPU.

Note: This method modifies the module in-place.

Returns: Module: self

Return type ~T

cuda(*device=None*)
Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Note: This method modifies the module in-place.

Args:

device (int, optional): if specified, all parameters will be copied to that device

Returns: Module: self

Return type ~T

property dither: *float*

Return type *float*

double()
Casts all floating point parameters and buffers to `double` datatype.

Note: This method modifies the module in-place.

Returns: Module: self

Return type ~T

dump_patches: bool = False

This allows better BC support for *load_state_dict()*. In *state_dict()*, the version number will be saved as in the attribute *_metadata* of the returned state dict, and thus pickled. *_metadata* is a dictionary with keys that follow the naming convention of state dict. See `_load_from_state_dict` on how to use this information in loading.

If new parameters/buffers are added/removed from a module, this number shall be bumped, and the module's *_load_from_state_dict* method can compare the version number and do appropriate changes if the state dict is from before the change.

eval()

Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between *.eval()* and several similar mechanisms that may be confused with it.

Returns: Module: self

Return type ~T

extra_repr()

Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

Return type `str`

float()

Casts all floating point parameters and buffers to `float` datatype.

Note: This method modifies the module in-place.

Returns: Module: self

Return type ~T

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

Return type `Tensor`

property frame_length: *float*

Return type *float*

property frame_shift: *float*

Return type *float*

get_buffer(*target*)

Returns the buffer given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Args:

target: The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns: torch.Tensor: The buffer referenced by `target`

Raises:

AttributeError: If the target string references an invalid path or resolves to something that is not a buffer

Return type `Tensor`

get_extra_state()

Returns any extra state to include in the module's state_dict. Implement this and a corresponding *set_extra_state()* for your module if you need to store extra state. This function is called when building the module's *state_dict()*.

Note that extra state should be pickleable to ensure working serialization of the state_dict. We only provide provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

Returns: object: Any extra state to store in the module's state_dict

Return type `Any`

get_parameter(*target*)

Returns the parameter given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Args:

target: The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns: torch.nn.Parameter: The Parameter referenced by `target`

Raises:

AttributeError: If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

Return type `Parameter`

get_submodule(*target*)

Returns the submodule given by `target` if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module A` that looks like this:

(The diagram shows an `nn.Module` `A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in `target`. A query against `named_modules` achieves the same result, but it is O(N) in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

Args:

target: The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

Returns: torch.nn.Module: The submodule referenced by `target`

Raises:

AttributeError: If the target string references an invalid path or resolves to something that is not an `nn.Module`

Return type `Module`

`half()`

Casts all floating point parameters and buffers to `half` datatype.

Note: This method modifies the module in-place.

Returns: Module: self

Return type ~T

`load_state_dict`(*state_dict*, *strict=True*)

Copies parameters and buffers from *`state_dict`* into this module and its descendants. If `strict` is `True`, then the keys of *`state_dict`* must exactly match the keys returned by this module's `state_dict()` function.

Args:

state_dict (dict): a dict containing parameters and persistent buffers.

strict (bool, optional): whether to strictly enforce that the keys in *`state_dict`* match the keys returned by this module's `state_dict()` function. Default: `True`

Returns:

`NamedTuple` with `missing_keys` and `unexpected_keys` fields:

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

Note: If a parameter or buffer is registered as `None` and its corresponding key exists in *`state_dict`*, *`load_state_dict()`* will raise a `RuntimeError`.

`modules()`

Returns an iterator over all modules in the network.

Yields: Module: a module in the network

Note: Duplicate modules are returned only once. In the following example, l will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
        print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

Return type `Iterator[Module]`

named_buffers(*prefix=''*, *recurse=True*)

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Args: prefix (str): prefix to prepend to all buffer names. recurse (bool): if True, then yields buffers of this module

and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields: (string, torch.Tensor): Tuple containing the name and buffer

Example:

```
>>> for name, buf in self.named_buffers():
>>>    if name in ['running_var']:
>>>        print(buf.size())
```

Return type `Iterator[Tuple[str, Tensor]]`

named_children()

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields: (string, Module): Tuple containing a name and child module

Example:

```
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

Return type `Iterator[Tuple[str, Module]]`

named_modules(*memo=None*, *prefix=''*, *remove_duplicate=True*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Args: memo: a memo to store the set of modules already added to the result prefix: a prefix that will be added to the name of the module remove_duplicate: whether to remove the duplicated module instances in the result

or not

Yields: (string, Module): Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
        print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

named_parameters(*prefix=''*, *recurse=True*)

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Args: prefix (str): prefix to prepend to all parameter names. recurse (bool): if True, then yields parameters of this module

and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: (string, Parameter): Tuple containing the name and parameter

Example:

```
>>> for name, param in self.named_parameters():
>>>    if name in ['bias']:
>>>        print(param.size())
```

Return type `Iterator[Tuple[str, Parameter]]`

online_inference(*x*, *context=None*)

Return type `Tuple[Tensor, Tensor]`

parameters(*recurse=True*)

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Args:

recurse (bool): if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: Parameter: module parameter

Example:

```
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

Return type Iterator[Parameter]

property preemph_coeff: *float*

Return type *float*

register_backward_hook(*hook*)

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

Return type RemovableHandle

register_buffer(*name*, *tensor*, *persistent=True*)

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's *state_dict*.

Buffers can be accessed as attributes using given names.

Args:

name (string): name of the buffer. The buffer can be accessed from this module using the given name

tensor (Tensor or None): buffer to be registered. If None, then operations that run on buffers, such as *cuda*, are ignored. If `None`, the buffer is **not** included in the module's *state_dict*.

persistent (bool): whether the buffer is part of this module's *state_dict*.

Example:

```
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

Return type None

register_forward_hook(*hook*)

Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after *forward()* is called.

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `RemovableHandle`

register_forward_pre_hook(*hook*)
Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `RemovableHandle`

register_full_backward_hook(*hook*)
Registers a backward hook on the module.

The hook will be called every time the gradients with respect to module inputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The `grad_input` and `grad_output` are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in `grad_input` and `grad_output` will be `None` for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module's forward function.

Warning: Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `RemovableHandle`

register_module(*name*, *module*)
Alias for *add_module()*.

Return type `None`

register_parameter(*name*, *param*)
Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Args:

name (string): name of the parameter. The parameter can be accessed from this module using the given name

param (Parameter or None): parameter to be added to the module. If `None`, then operations that run on parameters, such as `cuda`, are ignored. If `None`, the parameter is **not** included in the module's `state_dict`.

Return type `None`

property remove_dc_offset: bool

Return type `bool`

requires_grad_(*requires_grad=True*)
Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See locally-disable-grad-doc for a comparison between *.requires_grad_()* and several similar mechanisms that may be confused with it.

Args:

requires_grad (bool): whether autograd should record operations on parameters in this module. Default: `True`.

Returns: Module: self

Return type ~T

property sampling_rate: int

Return type `int`

set_extra_state(*state*)
This function is called from `load_state_dict()` to handle any extra state found within the *state_dict*. Implement this function and a corresponding `get_extra_state()` for your module if you need to store extra state within its *state_dict*.

Args: state (dict): Extra state from the *state_dict*

share_memory()
See `torch.Tensor.share_memory_()`

Return type ~T

state_dict(*destination=None*, *prefix=''*, *keep_vars=False*)
Returns a dictionary containing a whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to `None` are not included.

Returns:

dict: a dictionary containing a whole state of the module

Example:

```
>>> module.state_dict().keys()
['bias', 'weight']
```

`to`(**args*, ***kwargs*)

Moves and/or casts the parameters and buffers.

This can be called as

`to`(*device=None*, *dtype=None*, *non_blocking=False*)

`to`(*dtype*, *non_blocking=False*)

`to`(*tensor*, *non_blocking=False*)

`to`(*memory_format=torch.channels_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex `dtypes`. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Args:

device (`torch.device`): the desired device of the parameters and buffers in this module

dtype (`torch.dtype`): the desired floating point or complex dtype of the parameters and buffers in this module

tensor (torch.Tensor): Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module

memory_format (`torch.memory_format`): the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns: Module: self

Examples:

```
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
```

(continues on next page)

(continued from previous page)

```
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[ 0.3741+0.j,  0.2382+0.j],
        [ 0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)
```

to_empty(*, *device*)

Moves the parameters and buffers to the specified device without copying storage.

Args:

device (`torch.device`): The desired device of the parameters and buffers in this module.

Returns: Module: self

Return type ~T

train(*mode=True*)

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

Args:

mode (bool): whether to set training mode (`True`) or evaluation mode (`False`). Default: `True`.

Returns: Module: self

Return type ~T

type(*dst_type*)

Casts all parameters and buffers to `dst_type`.

Note: This method modifies the module in-place.

Args: dst_type (type or string): the desired type

Returns: Module: self

Return type ~T

property window_type: str

Return type `str`

xpu(*device=None*)

Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

Note: This method modifies the module in-place.

Arguments:

device (int, optional): if specified, all parameters will be copied to that device

Returns: Module: self

Return type ~T

zero_grad(*set_to_none=False*)

Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

Args:

set_to_none (bool): instead of setting to zero, set the grads to None. See `torch.optim.Optimizer.zero_grad()` for details.

Return type `None`

training: bool

class lhotse.features.kaldi.layers.**Wav2LogSpec**(*sampling_rate=16000*, *frame_length=0.025*, *frame_shift=0.01*, *round_to_power_of_two=True*, *remove_dc_offset=True*, *preemph_coeff=0.97*, *window_type='povey'*, *dither=0.0*, *snip_edges=False*, *energy_floor=1e-10*, *raw_energy=True*, *use_energy=True*, *use_fft_mag=False*)

Apply standard Kaldi preprocessing (dithering, removing DC offset, pre-emphasis, etc.) on the input waveforms and compute their Short-Time Fourier Transform (STFT). The STFT is transformed either to a log-magnitude spectrum (`use_fft_mag=True`) or a log-power spectrum (`use_fft_mag=False`).

Example:

```
>>> x = torch.randn(1, 16000, dtype=torch.float32)
>>> x.shape
torch.Size([1, 16000])
>>> t = Wav2LogSpec()
>>> t(x).shape
torch.Size([1, 100, 257])
```

The input is a tensor of shape `(batch_size, num_samples)`. The output is a tensor of shape `(batch_size, num_frames, num_fft_bins)`.

`__init__`(*sampling_rate=16000*, *frame_length=0.025*, *frame_shift=0.01*, *round_to_power_of_two=True*, *remove_dc_offset=True*, *preemph_coeff=0.97*, *window_type='povey'*, *dither=0.0*, *snip_edges=False*, *energy_floor=1e-10*, *raw_energy=True*, *use_energy=True*, *use_fft_mag=False*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

`T_destination`

alias of TypeVar('T_destination', bound=`Mapping[str, torch.Tensor]`)

`add_module`(*name*, *module*)

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Args:

name (string): name of the child module. The child module can be accessed from this module using the given name

module (Module): child module to be added to the module.

Return type `None`

`apply`(*fn*)

Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also nn-init-doc).

Args: fn (`Module` -> None): function to be applied to each submodule

Returns: Module: self

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

Return type ~T

bfloat16()
Casts all floating point parameters and buffers to `bfloat16` datatype.

Note: This method modifies the module in-place.

Returns: Module: self

Return type ~T

buffers(*recurse=True*)
Returns an iterator over module buffers.

Args:

recurse (bool): if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields: torch.Tensor: module buffer

Example:

```
>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

Return type `Iterator[Tensor]`

children()
Returns an iterator over immediate children modules.

Yields: Module: a child module

Return type `Iterator[Module]`

cpu()
Moves all model parameters and buffers to the CPU.

Note: This method modifies the module in-place.

Returns: Module: self

Return type ~T

cuda(*device=None*)
Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Note: This method modifies the module in-place.

Args:

device (int, optional): if specified, all parameters will be copied to that device

Returns: Module: self

Return type ~T

property dither: *float*

Return type *float*

double()

Casts all floating point parameters and buffers to `double` datatype.

Note: This method modifies the module in-place.

Returns: Module: self

Return type ~T

dump_patches: bool = False

This allows better BC support for `load_state_dict()`. In `state_dict()`, the version number will be saved as in the attribute *_metadata* of the returned state dict, and thus pickled. *_metadata* is a dictionary with keys that follow the naming convention of state dict. See `_load_from_state_dict` on how to use this information in loading.

If new parameters/buffers are added/removed from a module, this number shall be bumped, and the module's *_load_from_state_dict* method can compare the version number and do appropriate changes if the state dict is from before the change.

eval()

Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between *.eval()* and several similar mechanisms that may be confused with it.

Returns: Module: self

Return type ~T

extra_repr()

Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

Return type `str`

float()
Casts all floating point parameters and buffers to `float` datatype.

Note: This method modifies the module in-place.

Returns: Module: self

Return type ~T

forward(*x*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

Return type `Tensor`

property frame_length: *float*

Return type *float*

property frame_shift: *float*

Return type *float*

get_buffer(*target*)
Returns the buffer given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Args:

target: The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns: torch.Tensor: The buffer referenced by `target`

Raises:

AttributeError: If the target string references an invalid path or resolves to something that is not a buffer

Return type `Tensor`

get_extra_state()
Returns any extra state to include in the module's state_dict. Implement this and a corresponding *set_extra_state()* for your module if you need to store extra state. This function is called when building the module's *state_dict()*.

Note that extra state should be pickleable to ensure working serialization of the state_dict. We only provide provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

Returns: object: Any extra state to store in the module's state_dict

Return type Any

`get_parameter`(*target*)

Returns the parameter given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Args:

target: The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns: torch.nn.Parameter: The Parameter referenced by `target`

Raises:

AttributeError: If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

Return type Parameter

`get_submodule`(*target*)

Returns the submodule given by `target` if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module` `A` that looks like this:

(The diagram shows an `nn.Module` `A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in `target`. A query against `named_modules` achieves the same result, but it is O(N) in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

Args:

target: The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

Returns: torch.nn.Module: The submodule referenced by `target`

Raises:

AttributeError: If the target string references an invalid path or resolves to something that is not an `nn.Module`

Return type Module

`half`()

Casts all floating point parameters and buffers to `half` datatype.

Note: This method modifies the module in-place.

Returns: Module: self

Return type ~T

load_state_dict(*state_dict*, *strict=True*)

Copies parameters and buffers from `state_dict` into this module and its descendants. If `strict` is `True`, then the keys of `state_dict` must exactly match the keys returned by this module's `state_dict()` function.

Args:

state_dict (dict): a dict containing parameters and persistent buffers.

strict (bool, optional): whether to strictly enforce that the keys in `state_dict` match the keys returned by this module's `state_dict()` function. Default: `True`

Returns:

`NamedTuple` with `missing_keys` and `unexpected_keys` fields:

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

Note: If a parameter or buffer is registered as `None` and its corresponding key exists in `state_dict`, `load_state_dict()` will raise a `RuntimeError`.

modules()

Returns an iterator over all modules in the network.

Yields: Module: a module in the network

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
        print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

Return type `Iterator[Module]`

named_buffers(*prefix=''*, *recurse=True*)

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Args: prefix (str): prefix to prepend to all buffer names. recurse (bool): if True, then yields buffers of this module

and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields: (string, torch.Tensor): Tuple containing the name and buffer

Example:

```
>>> for name, buf in self.named_buffers():
>>>    if name in ['running_var']:
>>>        print(buf.size())
```

Return type `Iterator[Tuple[str, Tensor]]`

named_children()

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields: (string, Module): Tuple containing a name and child module

Example:

```
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

Return type `Iterator[Tuple[str, Module]]`

named_modules(*memo=None*, *prefix=''*, *remove_duplicate=True*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Args: memo: a memo to store the set of modules already added to the result prefix: a prefix that will be added to the name of the module remove_duplicate: whether to remove the duplicated module instances in the result

or not

Yields: (string, Module): Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
        print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

named_parameters(*prefix=''*, *recurse=True*)

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Args: prefix (str): prefix to prepend to all parameter names. recurse (bool): if True, then yields parameters of this module

and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: (string, Parameter): Tuple containing the name and parameter

Example:

```
>>> for name, param in self.named_parameters():
>>>    if name in ['bias']:
>>>        print(param.size())
```

Return type Iterator[Tuple[str, Parameter]]

online_inference(*x*, *context=None*)

Return type Tuple[Tensor, Tensor]

parameters(*recurse=True*)

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Args:

recurse (bool): if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: Parameter: module parameter

Example:

```
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

Return type Iterator[Parameter]

property preemph_coeff: *float*

Return type *float*

register_backward_hook(*hook*)

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

Return type RemovableHandle

register_buffer(*name*, *tensor*, *persistent=True*)

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's *state_dict*.

Buffers can be accessed as attributes using given names.

Args:

name (string): name of the buffer. The buffer can be accessed from this module using the given name

tensor (Tensor or None): buffer to be registered. If None, then operations that run on buffers, such as *cuda*, are ignored. If None, the buffer is **not** included in the module's *state_dict*.

persistent (bool): whether the buffer is part of this module's *state_dict*.

Example:

```
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

Return type None

register_forward_hook(*hook*)

Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the forward. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after *forward()* is called.

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling handle.remove()

Return type RemovableHandle

register_forward_pre_hook(*hook*)

Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the forward. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling handle.remove()

Return type RemovableHandle

register_full_backward_hook(*hook*)

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to module inputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The `grad_input` and `grad_output` are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in `grad_input` and `grad_output` will be `None` for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module's forward function.

> **Warning:** Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

Returns:

`torch.utils.hooks.RemovableHandle`: a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `RemovableHandle`

`register_module`(*name*, *module*)

Alias for `add_module()`.

Return type `None`

`register_parameter`(*name*, *param*)

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Args:

name (string): name of the parameter. The parameter can be accessed from this module using the given name

param (Parameter or None): parameter to be added to the module. If `None`, then operations that run on parameters, such as `cuda`, are ignored. If `None`, the parameter is **not** included in the module's `state_dict`.

Return type `None`

`property remove_dc_offset: bool`

Return type `bool`

`requires_grad_`(*requires_grad=True*)

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See locally-disable-grad-doc for a comparison between *.requires_grad_()* and several similar mechanisms that may be confused with it.

Args:

requires_grad (bool): whether autograd should record operations on parameters in this module. Default: `True`.

Returns: Module: self

Return type ~T

property sampling_rate: int

Return type `int`

set_extra_state(*state*)

This function is called from `load_state_dict()` to handle any extra state found within the *state_dict*. Implement this function and a corresponding `get_extra_state()` for your module if you need to store extra state within its *state_dict*.

Args: state (dict): Extra state from the *state_dict*

share_memory()

See `torch.Tensor.share_memory_()`

Return type ~T

state_dict(*destination=None*, *prefix=''*, *keep_vars=False*)

Returns a dictionary containing a whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to `None` are not included.

Returns:

dict: a dictionary containing a whole state of the module

Example:

```
>>> module.state_dict().keys()
['bias', 'weight']
```

to(**args*, ***kwargs*)

Moves and/or casts the parameters and buffers.

This can be called as

to(*device=None*, *dtype=None*, *non_blocking=False*)

to(*dtype*, *non_blocking=False*)

to(*tensor*, *non_blocking=False*)

to(*memory_format=torch.channels_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex `dtypes`. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Args:

device (`torch.device`): the desired device of the parameters and buffers in this module

dtype (`torch.dtype`): the desired floating point or complex dtype of the parameters and buffers in this module

tensor (torch.Tensor): Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module

memory_format (`torch.memory_format`): the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns: Module: self

Examples:

```
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[ 0.3741+0.j,  0.2382+0.j],
        [ 0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)
```

`to_empty`(*, *device*)
Moves the parameters and buffers to the specified device without copying storage.

Args:

device (`torch.device`): The desired device of the parameters and buffers in this module.

Returns: Module: self

Return type ~T

`train`(*mode=True*)
Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

Args:

mode (bool): whether to set training mode (`True`) or evaluation mode (`False`). Default: `True`.

Returns: Module: self

Return type ~T

`type`(*dst_type*)
Casts all parameters and buffers to `dst_type`.

Note: This method modifies the module in-place.

Args: dst_type (type or string): the desired type

Returns: Module: self

Return type ~T

`property window_type: str`

Return type `str`

`xpu`(*device=None*)
Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

Note: This method modifies the module in-place.

Arguments:

device (int, optional): if specified, all parameters will be copied to that device

Returns: Module: self

Return type ~T

zero_grad(*set_to_none=False*)
Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

Args:

set_to_none (bool): instead of setting to zero, set the grads to None. See `torch.optim.Optimizer.zero_grad()` for details.

Return type None

training: bool

class lhotse.features.kaldi.layers.**Wav2LogFilterBank**(*sampling_rate=16000, frame_length=0.025, frame_shift=0.01, round_to_power_of_two=True, remove_dc_offset=True, preemph_coeff=0.97, window_type='povey', dither=0.0, snip_edges=False, energy_floor=1e-10, raw_energy=True, use_energy=False, use_fft_mag=False, low_freq=20.0, high_freq=- 400.0, num_filters=80, norm_filters=False, torchaudio_compatible_mel_scale=True*)
Apply standard Kaldi preprocessing (dithering, removing DC offset, pre-emphasis, etc.) on the input waveforms and compute their log-Mel filter bank energies (also known as "fbank").

Example:

```
>>> x = torch.randn(1, 16000, dtype=torch.float32)
>>> x.shape
torch.Size([1, 16000])
>>> t = Wav2LogFilterBank()
>>> t(x).shape
torch.Size([1, 100, 80])
```

The input is a tensor of shape `(batch_size, num_samples)`. The output is a tensor of shape `(batch_size, num_frames, num_filters)`.

__init__(*sampling_rate=16000, frame_length=0.025, frame_shift=0.01, round_to_power_of_two=True, remove_dc_offset=True, preemph_coeff=0.97, window_type='povey', dither=0.0, snip_edges=False, energy_floor=1e-10, raw_energy=True, use_energy=False, use_fft_mag=False, low_freq=20.0, high_freq=- 400.0, num_filters=80, norm_filters=False, torchaudio_compatible_mel_scale=True*)
Initializes internal Module state, shared by both nn.Module and ScriptModule.

T_destination
alias of TypeVar('T_destination', bound=`Mapping[str, torch.Tensor]`)

add_module(*name, module*)
Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Args:

name (string): name of the child module. The child module can be accessed from this module using the given name

module (Module): child module to be added to the module.

Return type None

`apply(fn)`

Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also nn-init-doc).

Args: fn (`Module` -> None): function to be applied to each submodule

Returns: Module: self

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

Return type ~T

`bfloat16()`

Casts all floating point parameters and buffers to `bfloat16` datatype.

Note: This method modifies the module in-place.

Returns: Module: self

Return type ~T

`buffers(recurse=True)`

Returns an iterator over module buffers.

Args:

recurse (bool): if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields: torch.Tensor: module buffer

Example:

```
>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

Return type `Iterator[Tensor]`

children()

Returns an iterator over immediate children modules.

Yields: Module: a child module

Return type `Iterator[Module]`

cpu()

Moves all model parameters and buffers to the CPU.

Note: This method modifies the module in-place.

Returns: Module: self

Return type ~T

cuda(*device=None*)

Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Note: This method modifies the module in-place.

Args:

device (int, optional): if specified, all parameters will be copied to that device

Returns: Module: self

Return type ~T

property dither: *float*

Return type *float*

double()

Casts all floating point parameters and buffers to `double` datatype.

Note: This method modifies the module in-place.

Returns: Module: self

Return type ~T

dump_patches: bool = False

This allows better BC support for *load_state_dict()*. In *state_dict()*, the version number will be saved as in the attribute *_metadata* of the returned state dict, and thus pickled. *_metadata* is a dictionary with keys that follow the naming convention of state dict. See `_load_from_state_dict` on how to use this information in loading.

If new parameters/buffers are added/removed from a module, this number shall be bumped, and the module's *_load_from_state_dict* method can compare the version number and do appropriate changes if the state dict is from before the change.

eval()

Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between *.eval()* and several similar mechanisms that may be confused with it.

Returns: Module: self

Return type ~T

extra_repr()

Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

Return type `str`

float()

Casts all floating point parameters and buffers to `float` datatype.

Note: This method modifies the module in-place.

Returns: Module: self

Return type ~T

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

Return type `Tensor`

property frame_length: *float*

Return type *float*

property frame_shift: *float*

Return type *float*

get_buffer(*target*)

Returns the buffer given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Args:

target: The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns: torch.Tensor: The buffer referenced by `target`

Raises:

AttributeError: If the target string references an invalid path or resolves to something that is not a buffer

Return type `Tensor`

get_extra_state()

Returns any extra state to include in the module's state_dict. Implement this and a corresponding *set_extra_state()* for your module if you need to store extra state. This function is called when building the module's *state_dict()*.

Note that extra state should be pickleable to ensure working serialization of the state_dict. We only provide provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

Returns: object: Any extra state to store in the module's state_dict

Return type `Any`

get_parameter(*target*)

Returns the parameter given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Args:

target: The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns: torch.nn.Parameter: The Parameter referenced by `target`

Raises:

AttributeError: If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

Return type `Parameter`

get_submodule(*target*)

Returns the submodule given by `target` if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module A` that looks like this:

(The diagram shows an `nn.Module` `A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in `target`. A query against `named_modules` achieves the same result, but it is O(N) in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

Args:

target: The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

Returns: torch.nn.Module: The submodule referenced by `target`

Raises:

AttributeError: If the target string references an invalid path or resolves to something that is not an `nn.Module`

Return type `Module`

`half()`
Casts all floating point parameters and buffers to `half` datatype.

Note: This method modifies the module in-place.

Returns: Module: self

Return type ~T

`load_state_dict`(*state_dict*, *strict=True*)
Copies parameters and buffers from *`state_dict`* into this module and its descendants. If `strict` is `True`, then the keys of *`state_dict`* must exactly match the keys returned by this module's `state_dict()` function.

Args:

state_dict (dict): a dict containing parameters and persistent buffers.

strict (bool, optional): whether to strictly enforce that the keys in *`state_dict`* match the keys returned by this module's `state_dict()` function. Default: `True`

Returns:

`NamedTuple` with `missing_keys` and `unexpected_keys` fields:

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

Note: If a parameter or buffer is registered as `None` and its corresponding key exists in *`state_dict`*, *`load_state_dict()`* will raise a `RuntimeError`.

`modules()`
Returns an iterator over all modules in the network.

Yields: Module: a module in the network

Note: Duplicate modules are returned only once. In the following example, l will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
        print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

Return type `Iterator[Module]`

named_buffers(*prefix=''*, *recurse=True*)

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Args: prefix (str): prefix to prepend to all buffer names. recurse (bool): if True, then yields buffers of this module

and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields: (string, torch.Tensor): Tuple containing the name and buffer

Example:

```
>>> for name, buf in self.named_buffers():
>>>     if name in ['running_var']:
>>>         print(buf.size())
```

Return type `Iterator[Tuple[str, Tensor]]`

named_children()

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields: (string, Module): Tuple containing a name and child module

Example:

```
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

Return type `Iterator[Tuple[str, Module]]`

named_modules(*memo=None*, *prefix=''*, *remove_duplicate=True*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Args: memo: a memo to store the set of modules already added to the result prefix: a prefix that will be added to the name of the module remove_duplicate: whether to remove the duplicated module instances in the result

or not

Yields: (string, Module): Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
        print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

named_parameters(*prefix=''*, *recurse=True*)

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Args: prefix (str): prefix to prepend to all parameter names. recurse (bool): if True, then yields parameters of this module

and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: (string, Parameter): Tuple containing the name and parameter

Example:

```
>>> for name, param in self.named_parameters():
>>>    if name in ['bias']:
>>>        print(param.size())
```

Return type `Iterator[Tuple[str, Parameter]]`

online_inference(*x*, *context=None*)

Return type `Tuple[Tensor, Tensor]`

parameters(*recurse=True*)

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Args:

recurse (bool): if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: Parameter: module parameter

Example:

```
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

Return type Iterator[Parameter]

property preemph_coeff: *float*

Return type *float*

register_backward_hook(*hook*)

Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

Return type RemovableHandle

register_buffer(*name*, *tensor*, *persistent=True*)

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's *state_dict*.

Buffers can be accessed as attributes using given names.

Args:

name (string): name of the buffer. The buffer can be accessed from this module using the given name

tensor (Tensor or None): buffer to be registered. If None, then operations that run on buffers, such as *cuda*, are ignored. If `None`, the buffer is **not** included in the module's *state_dict*.

persistent (bool): whether the buffer is part of this module's *state_dict*.

Example:

```
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

Return type None

register_forward_hook(*hook*)

Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after *forward()* is called.

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `RemovableHandle`

register_forward_pre_hook(*hook*)
Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `RemovableHandle`

register_full_backward_hook(*hook*)
Registers a backward hook on the module.

The hook will be called every time the gradients with respect to module inputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The `grad_input` and `grad_output` are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in `grad_input` and `grad_output` will be `None` for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module's forward function.

Warning: Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `RemovableHandle`

register_module(*name*, *module*)
Alias for *add_module()*.

Return type `None`

register_parameter(*name*, *param*)

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Args:

name (string): name of the parameter. The parameter can be accessed from this module using the given name

param (Parameter or None): parameter to be added to the module. If `None`, then operations that run on parameters, such as `cuda`, are ignored. If `None`, the parameter is **not** included in the module's `state_dict`.

Return type `None`

property remove_dc_offset: bool

Return type `bool`

requires_grad_(*requires_grad=True*)

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See locally-disable-grad-doc for a comparison between *.requires_grad_()* and several similar mechanisms that may be confused with it.

Args:

requires_grad (bool): whether autograd should record operations on parameters in this module. Default: `True`.

Returns: Module: self

Return type ~T

property sampling_rate: int

Return type `int`

set_extra_state(*state*)

This function is called from `load_state_dict()` to handle any extra state found within the *state_dict*. Implement this function and a corresponding `get_extra_state()` for your module if you need to store extra state within its *state_dict*.

Args: state (dict): Extra state from the *state_dict*

share_memory()

See `torch.Tensor.share_memory_()`

Return type ~T

state_dict(*destination=None*, *prefix=''*, *keep_vars=False*)

Returns a dictionary containing a whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to `None` are not included.

Returns:

dict: a dictionary containing a whole state of the module

Example:

```
>>> module.state_dict().keys()
['bias', 'weight']
```

`to`(**args*, ***kwargs*)

Moves and/or casts the parameters and buffers.

This can be called as

`to`(*device=None*, *dtype=None*, *non_blocking=False*)

`to`(*dtype*, *non_blocking=False*)

`to`(*tensor*, *non_blocking=False*)

`to`(*memory_format=torch.channels_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex `dtypes`. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Args:

device (`torch.device`): the desired device of the parameters and buffers in this module

dtype (`torch.dtype`): the desired floating point or complex dtype of the parameters and buffers in this module

tensor (torch.Tensor): Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module

memory_format (`torch.memory_format`): the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns: Module: self

Examples:

```
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
```

(continues on next page)

(continued from previous page)

```
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)

>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[ 0.3741+0.j,  0.2382+0.j],
        [ 0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)
```

to_empty(*, *device*)

Moves the parameters and buffers to the specified device without copying storage.

Args:

device (torch.device): The desired device of the parameters and buffers in this module.

Returns: Module: self

Return type ~T

train(*mode=True*)

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

Args:

mode (bool): whether to set training mode (True) or evaluation mode (`False`). Default: `True`.

Returns: Module: self

Return type ~T

type(*dst_type*)

Casts all parameters and buffers to `dst_type`.

Note: This method modifies the module in-place.

Args: dst_type (type or string): the desired type

Returns: Module: self

Return type ~T

property window_type: str

Return type `str`

xpu(*device=None*)

Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

Note: This method modifies the module in-place.

Arguments:

device (int, optional): if specified, all parameters will be copied to that device

Returns: Module: self

Return type ~T

zero_grad(*set_to_none=False*)

Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

Args:

set_to_none (bool): instead of setting to zero, set the grads to None. See `torch.optim.Optimizer.zero_grad()` for details.

Return type `None`

training: bool

class lhotse.features.kaldi.layers.**Wav2MFCC**(*sampling_rate=16000, frame_length=0.025, frame_shift=0.01, round_to_power_of_two=True, remove_dc_offset=True, preemph_coeff=0.97, window_type='povey', dither=0.0, snip_edges=False, energy_floor=1e-10, raw_energy=True, use_energy=False, use_fft_mag=False, low_freq=20.0, high_freq=- 400.0, num_filters=23, norm_filters=False, num_ceps=13, cepstral_lifter=22, torchaudio_compatible_mel_scale=True*)

Apply standard Kaldi preprocessing (dithering, removing DC offset, pre-emphasis, etc.) on the input waveforms and compute their Mel-Frequency Cepstral Coefficients (MFCC).

Example:

```
>>> x = torch.randn(1, 16000, dtype=torch.float32)
>>> x.shape
torch.Size([1, 16000])
>>> t = Wav2MFCC()
```

(continues on next page)

(continued from previous page)

```
>>> t(x).shape
torch.Size([1, 100, 13])
```

The input is a tensor of shape `(batch_size, num_samples)`. The output is a tensor of shape `(batch_size, num_frames, num_ceps)`.

`__init__`(*sampling_rate=16000*, *frame_length=0.025*, *frame_shift=0.01*, *round_to_power_of_two=True*, *remove_dc_offset=True*, *preemph_coeff=0.97*, *window_type='povey'*, *dither=0.0*, *snip_edges=False*, *energy_floor=1e-10*, *raw_energy=True*, *use_energy=False*, *use_fft_mag=False*, *low_freq=20.0*, *high_freq=- 400.0*, *num_filters=23*, *norm_filters=False*, *num_ceps=13*, *cepstral_lifter=22*, *torchaudio_compatible_mel_scale=True*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

static `make_lifter`(*N*, *Q*)

Makes the liftering function

Args: N: Number of cepstral coefficients. Q: Liftering parameter

Returns: Liftering vector.

static `make_dct_matrix`(*num_ceps*, *num_filters*)

`T_destination`

alias of TypeVar('T_destination', bound=`Mapping[str, torch.Tensor]`)

`add_module`(*name*, *module*)

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Args:

name (string): name of the child module. The child module can be accessed from this module using the given name

module (Module): child module to be added to the module.

Return type `None`

`apply`(*fn*)

Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also nn-init-doc).

Args: fn (`Module` -> None): function to be applied to each submodule

Returns: Module: self

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
```

(continues on next page)

(continued from previous page)

```
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

Return type ~T

bfloat16()

Casts all floating point parameters and buffers to `bfloat16` datatype.

Note: This method modifies the module in-place.

Returns: Module: self

Return type ~T

buffers(*recurse=True*)

Returns an iterator over module buffers.

Args:

recurse (bool): if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields: torch.Tensor: module buffer

Example:

```
>>> for buf in model.buffers():
>>>     print(type(buf), buf.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

Return type `Iterator[Tensor]`

children()

Returns an iterator over immediate children modules.

Yields: Module: a child module

Return type `Iterator[Module]`

`cpu()`

Moves all model parameters and buffers to the CPU.

Note: This method modifies the module in-place.

Returns: Module: self

Return type ~T

`cuda(device=None)`

Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Note: This method modifies the module in-place.

Args:

device (int, optional): if specified, all parameters will be copied to that device

Returns: Module: self

Return type ~T

`property dither: float`

Return type *float*

`double()`

Casts all floating point parameters and buffers to `double` datatype.

Note: This method modifies the module in-place.

Returns: Module: self

Return type ~T

`dump_patches: bool = False`

This allows better BC support for `load_state_dict()`. In `state_dict()`, the version number will be saved as in the attribute *_metadata* of the returned state dict, and thus pickled. *_metadata* is a dictionary with keys that follow the naming convention of state dict. See `_load_from_state_dict` on how to use this information in loading.

If new parameters/buffers are added/removed from a module, this number shall be bumped, and the module's *_load_from_state_dict* method can compare the version number and do appropriate changes if the state dict is from before the change.

`eval()`

Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

This is equivalent with `self.train(False)`.

See locally-disable-grad-doc for a comparison between *.eval()* and several similar mechanisms that may be confused with it.

Returns: Module: self

Return type ~T

extra_repr()

Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

Return type `str`

float()

Casts all floating point parameters and buffers to `float` datatype.

Note: This method modifies the module in-place.

Returns: Module: self

Return type ~T

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

Return type `Tensor`

property frame_length: *float*

Return type *float*

property frame_shift: *float*

Return type *float*

get_buffer(*target*)

Returns the buffer given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Args:

target: The fully-qualified string name of the buffer to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns: torch.Tensor: The buffer referenced by `target`

Raises:

AttributeError: If the target string references an invalid path or resolves to something that is not a buffer

Return type Tensor

`get_extra_state()`

Returns any extra state to include in the module's state_dict. Implement this and a corresponding `set_extra_state()` for your module if you need to store extra state. This function is called when building the module's *state_dict()*.

Note that extra state should be pickleable to ensure working serialization of the state_dict. We only provide provide backwards compatibility guarantees for serializing Tensors; other objects may break backwards compatibility if their serialized pickled form changes.

Returns: object: Any extra state to store in the module's state_dict

Return type Any

`get_parameter(`*target*`)`

Returns the parameter given by `target` if it exists, otherwise throws an error.

See the docstring for `get_submodule` for a more detailed explanation of this method's functionality as well as how to correctly specify `target`.

Args:

target: The fully-qualified string name of the Parameter to look for. (See `get_submodule` for how to specify a fully-qualified string.)

Returns: torch.nn.Parameter: The Parameter referenced by `target`

Raises:

AttributeError: If the target string references an invalid path or resolves to something that is not an `nn.Parameter`

Return type Parameter

`get_submodule(`*target*`)`

Returns the submodule given by `target` if it exists, otherwise throws an error.

For example, let's say you have an `nn.Module` `A` that looks like this:

(The diagram shows an `nn.Module` `A`. `A` has a nested submodule `net_b`, which itself has two submodules `net_c` and `linear`. `net_c` then has a submodule `conv`.)

To check whether or not we have the `linear` submodule, we would call `get_submodule("net_b.linear")`. To check whether we have the `conv` submodule, we would call `get_submodule("net_b.net_c.conv")`.

The runtime of `get_submodule` is bounded by the degree of module nesting in `target`. A query against `named_modules` achieves the same result, but it is O(N) in the number of transitive modules. So, for a simple check to see if some submodule exists, `get_submodule` should always be used.

Args:

target: The fully-qualified string name of the submodule to look for. (See above example for how to specify a fully-qualified string.)

Returns: torch.nn.Module: The submodule referenced by `target`

Raises:

AttributeError: If the target string references an invalid path or resolves to something that is not an `nn.Module`

Return type `Module`

`half()`

Casts all floating point parameters and buffers to `half` datatype.

Note: This method modifies the module in-place.

Returns: Module: self

Return type ~T

`load_state_dict(state_dict, strict=True)`

Copies parameters and buffers from *state_dict* into this module and its descendants. If `strict` is `True`, then the keys of *state_dict* must exactly match the keys returned by this module's `state_dict()` function.

Args:

state_dict (dict): a dict containing parameters and persistent buffers.

strict (bool, optional): whether to strictly enforce that the keys in *state_dict* match the keys returned by this module's `state_dict()` function. Default: `True`

Returns:

`NamedTuple` with `missing_keys` and `unexpected_keys` fields:

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

Note: If a parameter or buffer is registered as `None` and its corresponding key exists in *state_dict*, *load_state_dict()* will raise a `RuntimeError`.

`modules()`

Returns an iterator over all modules in the network.

Yields: Module: a module in the network

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
        print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

Return type Iterator[Module]

named_buffers(*prefix=''*, *recurse=True*)

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Args: prefix (str): prefix to prepend to all buffer names. recurse (bool): if True, then yields buffers of this module

and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields: (string, torch.Tensor): Tuple containing the name and buffer

Example:

```
>>> for name, buf in self.named_buffers():
>>>    if name in ['running_var']:
>>>        print(buf.size())
```

Return type Iterator[Tuple[str, Tensor]]

named_children()

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields: (string, Module): Tuple containing a name and child module

Example:

```
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

Return type Iterator[Tuple[str, Module]]

named_modules(*memo=None*, *prefix=''*, *remove_duplicate=True*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Args: memo: a memo to store the set of modules already added to the result prefix: a prefix that will be added to the name of the module remove_duplicate: whether to remove the duplicated module instances in the result

or not

Yields: (string, Module): Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, l will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
        print(idx, '->', m)

0 -> ('', Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
```

(continues on next page)

(continued from previous page)

```
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

named_parameters(*prefix=''*, *recurse=True*)
Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Args: prefix (str): prefix to prepend to all parameter names. recurse (bool): if True, then yields parameters of this module

and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: (string, Parameter): Tuple containing the name and parameter

Example:

```
>>> for name, param in self.named_parameters():
>>>    if name in ['bias']:
>>>        print(param.size())
```

Return type Iterator[Tuple[str, Parameter]]

online_inference(*x*, *context=None*)

Return type Tuple[Tensor, Tensor]

parameters(*recurse=True*)
Returns an iterator over module parameters.

This is typically passed to an optimizer.

Args:

recurse (bool): if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields: Parameter: module parameter

Example:

```
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

Return type Iterator[Parameter]

property preemph_coeff: *float*

Return type *float*

register_backward_hook(*hook*)
Registers a backward hook on the module.

This function is deprecated in favor of `register_full_backward_hook()` and the behavior of this function will change in future versions.

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `RemovableHandle`

register_buffer(*name*, *tensor*, *persistent=True*)

Adds a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's *state_dict*.

Buffers can be accessed as attributes using given names.

Args:

name (string): name of the buffer. The buffer can be accessed from this module using the given name

tensor (Tensor or None): buffer to be registered. If `None`, then operations that run on buffers, such as *cuda*, are ignored. If `None`, the buffer is **not** included in the module's *state_dict*.

persistent (bool): whether the buffer is part of this module's *state_dict*.

Example:

```
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

Return type `None`

register_forward_hook(*hook*)

Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None or modified output
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the output. It can modify the input inplace but it will not have effect on forward since this is called after *forward()* is called.

Returns:

torch.utils.hooks.RemovableHandle: a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `RemovableHandle`

register_forward_pre_hook(*hook*)

Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None or modified input
```

The input contains only the positional arguments given to the module. Keyword arguments won't be passed to the hooks and only to the `forward`. The hook can modify the input. User can either return a tuple or a single modified value in the hook. We will wrap the value into a tuple if a single value is returned(unless that value is already a tuple).

Returns:

`torch.utils.hooks.RemovableHandle`: a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `RemovableHandle`

`register_full_backward_hook`(*hook*)

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to module inputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> tuple(Tensor) or None
```

The `grad_input` and `grad_output` are tuples that contain the gradients with respect to the inputs and outputs respectively. The hook should not modify its arguments, but it can optionally return a new gradient with respect to the input that will be used in place of `grad_input` in subsequent computations. `grad_input` will only correspond to the inputs given as positional arguments and all kwarg arguments are ignored. Entries in `grad_input` and `grad_output` will be `None` for all non-Tensor arguments.

For technical reasons, when this hook is applied to a Module, its forward function will receive a view of each Tensor passed to the Module. Similarly the caller will receive a view of each Tensor returned by the Module's forward function.

Warning: Modifying inputs or outputs inplace is not allowed when using backward hooks and will raise an error.

Returns:

`torch.utils.hooks.RemovableHandle`: a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `RemovableHandle`

`register_module`(*name*, *module*)

Alias for `add_module()`.

Return type `None`

`register_parameter`(*name*, *param*)

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Args:

name (string): name of the parameter. The parameter can be accessed from this module using the given name

param (Parameter or None): parameter to be added to the module. If `None`, then operations that run on parameters, such as `cuda`, are ignored. If `None`, the parameter is **not** included in the module's `state_dict`.

Return type None

property remove_dc_offset: bool

Return type bool

requires_grad_(*requires_grad=True*)

Change if autograd should record operations on parameters in this module.

This method sets the parameters' `requires_grad` attributes in-place.

This method is helpful for freezing part of the module for finetuning or training parts of a model individually (e.g., GAN training).

See locally-disable-grad-doc for a comparison between *.requires_grad_()* and several similar mechanisms that may be confused with it.

Args:

requires_grad (bool): whether autograd should record operations on parameters in this module. Default: `True`.

Returns: Module: self

Return type ~T

property sampling_rate: int

Return type int

set_extra_state(*state*)

This function is called from `load_state_dict()` to handle any extra state found within the *state_dict*. Implement this function and a corresponding `get_extra_state()` for your module if you need to store extra state within its *state_dict*.

Args: state (dict): Extra state from the *state_dict*

share_memory()

See `torch.Tensor.share_memory_()`

Return type ~T

state_dict(*destination=None*, *prefix=''*, *keep_vars=False*)

Returns a dictionary containing a whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to `None` are not included.

Returns:

dict: a dictionary containing a whole state of the module

Example:

```
>>> module.state_dict().keys()
['bias', 'weight']
```

to(**args*, ***kwargs*)

Moves and/or casts the parameters and buffers.

This can be called as

to(*device=None*, *dtype=None*, *non_blocking=False*)

to(*dtype*, *non_blocking=False*)

to(*tensor*, *non_blocking=False*)

to(*memory_format=torch.channels_last*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point or complex `dtypes`. In addition, this method will only cast the floating point or complex parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Args:

device (`torch.device`): the desired device of the parameters and buffers in this module

dtype (`torch.dtype`): the desired floating point or complex dtype of the parameters and buffers in this module

tensor (torch.Tensor): Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module

memory_format (`torch.memory_format`): the desired memory format for 4D parameters and buffers in this module (keyword only argument)

Returns: Module: self

Examples:

```
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)
```

(continues on next page)

(continued from previous page)

```
>>> linear = nn.Linear(2, 2, bias=None).to(torch.cdouble)
>>> linear.weight
Parameter containing:
tensor([[ 0.3741+0.j,  0.2382+0.j],
        [ 0.5593+0.j, -0.4443+0.j]], dtype=torch.complex128)
>>> linear(torch.ones(3, 2, dtype=torch.cdouble))
tensor([[0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j],
        [0.6122+0.j, 0.1150+0.j]], dtype=torch.complex128)
```

to_empty(*, *device*)

Moves the parameters and buffers to the specified device without copying storage.

Args:

device (`torch.device`): The desired device of the parameters and buffers in this module.

Returns: Module: self

Return type ~T

train(*mode=True*)

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

Args:

mode (bool): whether to set training mode (`True`) or evaluation mode (`False`). Default: `True`.

Returns: Module: self

Return type ~T

type(*dst_type*)

Casts all parameters and buffers to `dst_type`.

Note: This method modifies the module in-place.

Args: dst_type (type or string): the desired type

Returns: Module: self

Return type ~T

property window_type: str

Return type `str`

xpu(*device=None*)

Moves all model parameters and buffers to the XPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on XPU while being optimized.

Note: This method modifies the module in-place.

Arguments:

device (int, optional): if specified, all parameters will be copied to that device

Returns: Module: self

Return type ~T

`zero_grad`(*set_to_none=False*)

Sets gradients of all model parameters to zero. See similar function under `torch.optim.Optimizer` for more context.

Args:

set_to_none (bool): instead of setting to zero, set the grads to None. See `torch.optim.Optimizer.zero_grad()` for details.

Return type `None`

`training: bool`

`lhotse.features.kaldi.layers.create_mel_scale`(*num_filters, fft_length, sampling_rate, low_freq=0, high_freq=None, norm_filters=True*)

Return type `Tensor`

`lhotse.features.kaldi.layers.available_windows`()

Return type `List[str]`

`lhotse.features.kaldi.layers.create_frame_window`(*window_size, window_type='povey', blackman_coeff=0.42*)

Returns a window function with the given type and size

`lhotse.features.kaldi.layers.lin2mel`(*x*)

`lhotse.features.kaldi.layers.mel2lin`(*x*)

`lhotse.features.kaldi.layers.next_power_of_2`(*x*)

Returns the smallest power of 2 that is greater than x.

Original source: TorchAudio (torchaudio/compliance/kaldi.py)

Return type `int`

9.3.4 Torchaudio feature extractors

class lhotse.features.fbank.**TorchaudioFbankConfig**(*dither=0.0*, *window_type='povey'*, *frame_length=0.025*, *frame_shift=0.01*, *remove_dc_offset=True*, *round_to_power_of_two=True*, *energy_floor=1e-10*, *min_duration=0.0*, *preemphasis_coefficient=0.97*, *raw_energy=True*, *low_freq=20.0*, *high_freq=- 400.0*, *num_mel_bins=80*, *use_energy=False*, *vtln_low=100.0*, *vtln_high=- 500.0*, *vtln_warp=1.0*)

dither: float = 0.0

window_type: str = 'povey'

frame_length: float = 0.025

frame_shift: float = 0.01

remove_dc_offset: bool = True

round_to_power_of_two: bool = True

energy_floor: float = 1e-10

min_duration: float = 0.0

preemphasis_coefficient: float = 0.97

raw_energy: bool = True

low_freq: float = 20.0

high_freq: float = -400.0

num_mel_bins: int = 80

use_energy: bool = False

vtln_low: float = 100.0

vtln_high: float = -500.0

vtln_warp: float = 1.0

to_dict()

Return type Dict[str, Any]

static from_dict(*data*)

Return type *TorchaudioFbankConfig*

__init__(*dither=0.0*, *window_type='povey'*, *frame_length=0.025*, *frame_shift=0.01*, *remove_dc_offset=True*, *round_to_power_of_two=True*, *energy_floor=1e-10*, *min_duration=0.0*, *preemphasis_coefficient=0.97*, *raw_energy=True*, *low_freq=20.0*, *high_freq=- 400.0*, *num_mel_bins=80*, *use_energy=False*, *vtln_low=100.0*, *vtln_high=- 500.0*, *vtln_warp=1.0*)

class lhotse.features.fbank.**TorchaudioFbank**(*config=None*)

Log Mel energy filter bank feature extractor based on `torchaudio.compliance.kaldi.fbank` function.

`name = 'fbank'`

`config_type`
alias of *lhotse.features.fbank.TorchaudioFbankConfig*

`feature_dim`(*sampling_rate*)

Return type `int`

`static mix`(*features_a*, *features_b*, *energy_scaling_factor_b*)
Perform feature-domain mix of two signals, `a` and `b`, and return the mixed signal.

Parameters

- **features_a** (`ndarray`) – Left-hand side (reference) signal.
- **features_b** (`ndarray`) – Right-hand side (mixed-in) signal.
- **energy_scaling_factor_b** (`float`) – A scaling factor for `features_b` energy. It is used to achieve a specific SNR. E.g. to mix with an SNR of 10dB when both `features_a` and `features_b` energies are 100, the `features_b` signal energy needs to be scaled by 0.1. Since different features (e.g. spectrogram, fbank, MFCC) require different combination of transformations (e.g. exp, log, sqrt, pow) to allow mixing of two signals, the exact place where to apply `energy_scaling_factor_b` to the signal is determined by the implementer.

Return type `ndarray`

Returns A mixed feature matrix.

`static compute_energy`(*features*)
Compute the total energy of a feature matrix. How the energy is computed depends on a particular type of features. It is expected that when implemented, `compute_energy` will never return zero.

Parameters **features** (`ndarray`) – A feature matrix.

Return type `float`

Returns A positive float value of the signal energy.

`__init__`(*config=None*)

`property device: Union[str, torch.device]`

Return type `Union[str, device]`

`extract`(*samples*, *sampling_rate*)
Defines how to extract features using a numpy ndarray of audio samples and the sampling rate.

Return type `ndarray`

Returns a numpy ndarray representing the feature matrix.

`extract_batch`(*samples*, *sampling_rate*)
Performs batch extraction. It is not guaranteed to be faster than `FeatureExtractor.extract()` – it depends on whether the implementation of a particular feature extractor supports accelerated batch computation.

Note: Unless overridden by child classes, it defaults to sequentially calling `FeatureExtractor.extract()` on the inputs.

Note: This method *should* support variable length inputs.

Return type Union[ndarray, Tensor, List[ndarray], List[Tensor]]

extract_from_recording_and_store(*recording*, *storage*, *offset=0*, *duration=None*, *channels=None*, *augment_fn=None*)

Extract the features from a Recording in a full pipeline:

- load audio from disk;
- optionally, perform audio augmentation;
- extract the features;
- save them to disk in a specified directory;
- return a Features object with a description of the extracted features and the source data used.

Parameters

- **recording** (*Recording*) – a Recording that specifies what's the input audio.
- **storage** (*FeaturesWriter*) – a FeaturesWriter object that will handle storing the feature matrices.
- **offset** (float) – an optional offset in seconds for where to start reading the recording.
- **duration** (Optional[float]) – an optional duration specifying how much audio to load from the recording.
- **channels** (Union[int, List[int], None]) – an optional int or list of ints, specifying the channels; by default, all channels will be used.
- **augment_fn** (Optional[Callable[[ndarray, int], ndarray]]) – an optional WavAugmenter instance to modify the waveform before feature extraction.

Return type *Features*

Returns a Features manifest item for the extracted feature matrix.

extract_from_samples_and_store(*samples*, *storage*, *sampling_rate*, *offset=0*, *channel=None*, *augment_fn=None*)

Extract the features from an array of audio samples in a full pipeline:

- optional audio augmentation;
- extract the features;
- save them to disk in a specified directory;
- return a Features object with a description of the extracted features.

Note, unlike in extract_from_recording_and_store, the returned Features object might not be suitable to store in a FeatureSet, as it does not reference any particular Recording. Instead, this method is useful when extracting features from cuts - especially MixedCut instances, which may be created from multiple recordings and channels.

Parameters

- **samples** (ndarray) – a numpy ndarray with the audio samples.
- **sampling_rate** (int) – integer sampling rate of samples.

- **storage** (*FeaturesWriter*) – a FeaturesWriter object that will handle storing the feature matrices.
- **offset** (float) – an offset in seconds for where to start reading the recording - when used for Cut feature extraction, must be equal to Cut.start.
- **channel** (Optional[int]) – an optional channel number to insert into Features manifest.
- **augment_fn** (Optional[Callable[[ndarray, int], ndarray]]) – an optional WavAugmenter instance to modify the waveform before feature extraction.

Return type *Features*

Returns a Features manifest item for the extracted feature matrix (it is not written to disk).

property frame_shift: float

Return type float

classmethod from_dict(*data*)

Return type *FeatureExtractor*

classmethod from_yaml(*path*)

Return type *FeatureExtractor*

to_dict()

Return type Dict[str, Any]

to_yaml(*path*)

class lhotse.features.mfcc.**TorchaudioMfccConfig**(*dither=0.0*, *window_type='povey'*, *frame_length=0.025*, *frame_shift=0.01*, *remove_dc_offset=True*, *round_to_power_of_two=True*, *energy_floor=1e-10*, *min_duration=0.0*, *preemphasis_coefficient=0.97*, *raw_energy=True*, *low_freq=20.0*, *high_freq=- 400.0*, *num_mel_bins=23*, *use_energy=False*, *vtln_low=100.0*, *vtln_high=- 500.0*, *vtln_warp=1.0*, *cepstral_lifter=22.0*, *num_ceps=13*)

dither: float = 0.0

window_type: str = 'povey'

frame_length: float = 0.025

frame_shift: float = 0.01

remove_dc_offset: bool = True

round_to_power_of_two: bool = True

energy_floor: float = 1e-10

min_duration: float = 0.0

preemphasis_coefficient: float = 0.97

raw_energy: bool = True

low_freq: float = 20.0

high_freq: float = -400.0

num_mel_bins: int = 23

use_energy: bool = False

vtln_low: float = 100.0

vtln_high: float = -500.0

vtln_warp: float = 1.0

cepstral_lifter: float = 22.0

num_ceps: int = 13

to_dict()

Return type `Dict[str, Any]`

static **from_dict**(*data*)

Return type *TorchaudioMfccConfig*

__init__(*dither=0.0*, *window_type='povey'*, *frame_length=0.025*, *frame_shift=0.01*, *remove_dc_offset=True*, *round_to_power_of_two=True*, *energy_floor=1e-10*, *min_duration=0.0*, *preemphasis_coefficient=0.97*, *raw_energy=True*, *low_freq=20.0*, *high_freq=- 400.0*, *num_mel_bins=23*, *use_energy=False*, *vtln_low=100.0*, *vtln_high=- 500.0*, *vtln_warp=1.0*, *cepstral_lifter=22.0*, *num_ceps=13*)

class lhotse.features.mfcc.**TorchaudioMfcc**(*config=None*)

MFCC feature extractor based on `torchaudio.compliance.kaldi.mfcc` function.

name = 'mfcc'

config_type

alias of *lhotse.features.mfcc.TorchaudioMfccConfig*

feature_dim(*sampling_rate*)

Return type `int`

__init__(*config=None*)

static **compute_energy**(*features*)

Compute the total energy of a feature matrix. How the energy is computed depends on a particular type of features. It is expected that when implemented, `compute_energy` will never return zero.

Parameters **features** (`ndarray`) – A feature matrix.

Return type `float`

Returns A positive float value of the signal energy.

property device: Union[str, torch.device]

Return type `Union[str, device]`

extract(*samples*, *sampling_rate*)

Defines how to extract features using a numpy ndarray of audio samples and the sampling rate.

Return type `ndarray`

Returns a numpy ndarray representing the feature matrix.

extract_batch(*samples*, *sampling_rate*)

Performs batch extraction. It is not guaranteed to be faster than `FeatureExtractor.extract()` – it depends on whether the implementation of a particular feature extractor supports accelerated batch computation.

Note: Unless overridden by child classes, it defaults to sequentially calling `FeatureExtractor.extract()` on the inputs.

Note: This method *should* support variable length inputs.

Return type `Union[ndarray, Tensor, List[ndarray], List[Tensor]]`

extract_from_recording_and_store(*recording*, *storage*, *offset=0*, *duration=None*, *channels=None*, *augment_fn=None*)

Extract the features from a `Recording` in a full pipeline:

- load audio from disk;
- optionally, perform audio augmentation;
- extract the features;
- save them to disk in a specified directory;
- return a `Features` object with a description of the extracted features and the source data used.

Parameters

- **recording** (*Recording*) – a `Recording` that specifies what's the input audio.
- **storage** (*FeaturesWriter*) – a `FeaturesWriter` object that will handle storing the feature matrices.
- **offset** (`float`) – an optional offset in seconds for where to start reading the recording.
- **duration** (`Optional[float]`) – an optional duration specifying how much audio to load from the recording.
- **channels** (`Union[int, List[int], None]`) – an optional int or list of ints, specifying the channels; by default, all channels will be used.
- **augment_fn** (`Optional[Callable[[ndarray, int], ndarray]]`) – an optional `WavAugmenter` instance to modify the waveform before feature extraction.

Return type *Features*

Returns a `Features` manifest item for the extracted feature matrix.

extract_from_samples_and_store(*samples*, *storage*, *sampling_rate*, *offset=0*, *channel=None*, *augment_fn=None*)

Extract the features from an array of audio samples in a full pipeline:

- optional audio augmentation;
- extract the features;

- save them to disk in a specified directory;
- return a `Features` object with a description of the extracted features.

Note, unlike in `extract_from_recording_and_store`, the returned `Features` object might not be suitable to store in a `FeatureSet`, as it does not reference any particular `Recording`. Instead, this method is useful when extracting features from cuts - especially `MixedCut` instances, which may be created from multiple recordings and channels.

Parameters

- **samples** (`ndarray`) – a numpy ndarray with the audio samples.
- **sampling_rate** (`int`) – integer sampling rate of `samples`.
- **storage** (*FeaturesWriter*) – a `FeaturesWriter` object that will handle storing the feature matrices.
- **offset** (`float`) – an offset in seconds for where to start reading the recording - when used for `Cut` feature extraction, must be equal to `Cut.start`.
- **channel** (`Optional[int]`) – an optional channel number to insert into `Features` manifest.
- **augment_fn** (`Optional[Callable[[ndarray, int], ndarray]]`) – an optional `WavAugmenter` instance to modify the waveform before feature extraction.

Return type *Features*

Returns a `Features` manifest item for the extracted feature matrix (it is not written to disk).

property frame_shift: float

Return type `float`

classmethod from_dict(*data*)

Return type *FeatureExtractor*

classmethod from_yaml(*path*)

Return type *FeatureExtractor*

static mix(*features_a*, *features_b*, *energy_scaling_factor_b*)

Perform feature-domain mix of two signals, `a` and `b`, and return the mixed signal.

Parameters

- **features_a** (`ndarray`) – Left-hand side (reference) signal.
- **features_b** (`ndarray`) – Right-hand side (mixed-in) signal.
- **energy_scaling_factor_b** (`float`) – A scaling factor for `features_b` energy. It is used to achieve a specific SNR. E.g. to mix with an SNR of 10dB when both `features_a` and `features_b` energies are 100, the `features_b` signal energy needs to be scaled by 0.1. Since different features (e.g. spectrogram, fbank, MFCC) require different combination of transformations (e.g. exp, log, sqrt, pow) to allow mixing of two signals, the exact place where to apply `energy_scaling_factor_b` to the signal is determined by the implementer.

Return type `ndarray`

Returns A mixed feature matrix.

to_dict()

Return type Dict[str, Any]

to_yaml(*path*)

class lhotse.features.spectrogram.**SpectrogramConfig**(*dither=0.0*, *window_type='povey'*, *frame_length=0.025*, *frame_shift=0.01*, *remove_dc_offset=True*, *round_to_power_of_two=True*, *energy_floor=1e-10*, *min_duration=0.0*, *preemphasis_coefficient=0.97*, *raw_energy=True*)

dither: float = 0.0

window_type: str = 'povey'

frame_length: float = 0.025

frame_shift: float = 0.01

remove_dc_offset: bool = True

round_to_power_of_two: bool = True

energy_floor: float = 1e-10

min_duration: float = 0.0

preemphasis_coefficient: float = 0.97

raw_energy: bool = True

to_dict()

Return type Dict[str, Any]

static **from_dict**(*data*)

Return type *SpectrogramConfig*

__init__(*dither=0.0*, *window_type='povey'*, *frame_length=0.025*, *frame_shift=0.01*, *remove_dc_offset=True*, *round_to_power_of_two=True*, *energy_floor=1e-10*, *min_duration=0.0*, *preemphasis_coefficient=0.97*, *raw_energy=True*)

class lhotse.features.spectrogram.**Spectrogram**(*config=None*)

Log spectrogram feature extractor based on torchaudio.compliance.kaldi.spectrogram function.

name = 'spectrogram'

config_type

alias of *lhotse.features.spectrogram.SpectrogramConfig*

feature_dim(*sampling_rate*)

Return type int

static **mix**(*features_a*, *features_b*, *energy_scaling_factor_b*)

Perform feature-domain mix of two signals, a and b, and return the mixed signal.

Parameters

- **features_a** (ndarray) – Left-hand side (reference) signal.
- **features_b** (ndarray) – Right-hand side (mixed-in) signal.
- **energy_scaling_factor_b** (float) – A scaling factor for features_b energy. It is used to achieve a specific SNR. E.g. to mix with an SNR of 10dB when both features_a and features_b energies are 100, the features_b signal energy needs to be scaled by 0.1. Since different features (e.g. spectrogram, fbank, MFCC) require different combination of transformations (e.g. exp, log, sqrt, pow) to allow mixing of two signals, the exact place where to apply energy_scaling_factor_b to the signal is determined by the implementer.

Return type ndarray

Returns A mixed feature matrix.

static **compute_energy**(*features*)
Compute the total energy of a feature matrix. How the energy is computed depends on a particular type of features. It is expected that when implemented, compute_energy will never return zero.

Parameters **features** (ndarray) – A feature matrix.

Return type float

Returns A positive float value of the signal energy.

__init__(*config=None*)

property **device**: Union[str, torch.device]

Return type Union[str, device]

extract(*samples*, *sampling_rate*)
Defines how to extract features using a numpy ndarray of audio samples and the sampling rate.

Return type ndarray

Returns a numpy ndarray representing the feature matrix.

extract_batch(*samples*, *sampling_rate*)
Performs batch extraction. It is not guaranteed to be faster than FeatureExtractor.extract() – it depends on whether the implementation of a particular feature extractor supports accelerated batch computation.

Note: Unless overridden by child classes, it defaults to sequentially calling FeatureExtractor.extract() on the inputs.

Note: This method *should* support variable length inputs.

Return type Union[ndarray, Tensor, List[ndarray], List[Tensor]]

extract_from_recording_and_store(*recording*, *storage*, *offset=0*, *duration=None*, *channels=None*, *augment_fn=None*)
Extract the features from a Recording in a full pipeline:

- load audio from disk;
- optionally, perform audio augmentation;

- extract the features;
- save them to disk in a specified directory;
- return a `Features` object with a description of the extracted features and the source data used.

Parameters

- **recording** (*Recording*) – a `Recording` that specifies what's the input audio.
- **storage** (*FeaturesWriter*) – a `FeaturesWriter` object that will handle storing the feature matrices.
- **offset** (`float`) – an optional offset in seconds for where to start reading the recording.
- **duration** (`Optional[float]`) – an optional duration specifying how much audio to load from the recording.
- **channels** (`Union[int, List[int], None]`) – an optional int or list of ints, specifying the channels; by default, all channels will be used.
- **augment_fn** (`Optional[Callable[[ndarray, int], ndarray]]`) – an optional `WavAugmenter` instance to modify the waveform before feature extraction.

Return type *Features*

Returns a `Features` manifest item for the extracted feature matrix.

extract_from_samples_and_store(*samples*, *storage*, *sampling_rate*, *offset=0*, *channel=None*, *augment_fn=None*)

Extract the features from an array of audio samples in a full pipeline:

- optional audio augmentation;
- extract the features;
- save them to disk in a specified directory;
- return a `Features` object with a description of the extracted features.

Note, unlike in `extract_from_recording_and_store`, the returned `Features` object might not be suitable to store in a `FeatureSet`, as it does not reference any particular `Recording`. Instead, this method is useful when extracting features from cuts - especially `MixedCut` instances, which may be created from multiple recordings and channels.

Parameters

- **samples** (`ndarray`) – a numpy ndarray with the audio samples.
- **sampling_rate** (`int`) – integer sampling rate of `samples`.
- **storage** (*FeaturesWriter*) – a `FeaturesWriter` object that will handle storing the feature matrices.
- **offset** (`float`) – an offset in seconds for where to start reading the recording - when used for `Cut` feature extraction, must be equal to `Cut.start`.
- **channel** (`Optional[int]`) – an optional channel number to insert into `Features` manifest.
- **augment_fn** (`Optional[Callable[[ndarray, int], ndarray]]`) – an optional `WavAugmenter` instance to modify the waveform before feature extraction.

Return type *Features*

Returns a `Features` manifest item for the extracted feature matrix (it is not written to disk).

property frame_shift: float

Return type float

classmethod from_dict(*data*)

Return type *FeatureExtractor*

classmethod from_yaml(*path*)

Return type *FeatureExtractor*

to_dict()

Return type Dict[str, Any]

to_yaml(*path*)

9.3.5 Librosa filter-bank

class lhotse.features.librosa_fbank.**LibrosaFbankConfig**(*sampling_rate=22050*, *fft_size=1024*, *hop_size=256*, *win_length=None*, *window='hann'*, *num_mel_bins=80*, *fmin=80*, *fmax=7600*)

Default librosa config with values consistent with various TTS projects.

This config is intended for use with popular TTS projects such as [ParallelWaveGAN](https://github.com/kan-bayashi/ParallelWaveGAN) Warning: You may need to normalize your features.

sampling_rate: int = 22050

fft_size: int = 1024

hop_size: int = 256

win_length: int = None

window: str = 'hann'

num_mel_bins: int = 80

fmin: int = 80

fmax: int = 7600

to_dict()

Return type Dict[str, Any]

static from_dict(*data*)

Return type *LibrosaFbankConfig*

__init__(*sampling_rate=22050*, *fft_size=1024*, *hop_size=256*, *win_length=None*, *window='hann'*, *num_mel_bins=80*, *fmin=80*, *fmax=7600*)

lhotse.features.librosa_fbank.**pad_or_truncate_features**(*feats*, *expected_num_frames*, *abs_tol=1*, *pad_value=- 23.025850929940457*)

lhotse.features.librosa_fbank.**logmelfilterbank**(*audio*, *sampling_rate*, *fft_size=1024*, *hop_size=256*, *win_length=None*, *window='hann'*, *num_mel_bins=80*, *fmin=80*, *fmax=7600*, *eps=1e-10*)

Compute log-Mel filterbank feature.

Args: audio (ndarray): Audio signal (T,). sampling_rate (int): Sampling rate. fft_size (int): FFT size. hop_size (int): Hop size. win_length (int): Window length. If set to None, it will be the same as fft_size. window (str): Window function type. num_mel_bins (int): Number of mel basis. fmin (int): Minimum frequency in mel basis calculation. fmax (int): Maximum frequency in mel basis calculation. eps (float): Epsilon value to avoid inf in log calculation.

Returns: ndarray: Log Mel filterbank feature (#source_feats, num_mel_bins).

class lhotse.features.librosa_fbank.**LibrosaFbank**(*config=None*)

Librosa fbank feature extractor

Differs from Fbank extractor in that it uses librosa backend for stft and mel scale calculations. It can be easily configured to be compatible with existing speech-related projects that use librosa features.

name = 'librosa-fbank'

config_type

alias of *lhotse.features.librosa_fbank.LibrosaFbankConfig*

property frame_shift: float

Return type float

feature_dim(*sampling_rate*)

Return type int

extract(*samples*, *sampling_rate*)

Defines how to extract features using a numpy ndarray of audio samples and the sampling rate.

Return type ndarray

Returns a numpy ndarray representing the feature matrix.

static **mix**(*features_a*, *features_b*, *energy_scaling_factor_b*)

Perform feature-domain mix of two signals, a and b, and return the mixed signal.

Parameters

- **features_a** (ndarray) – Left-hand side (reference) signal.
- **features_b** (ndarray) – Right-hand side (mixed-in) signal.
- **energy_scaling_factor_b** (float) – A scaling factor for features_b energy. It is used to achieve a specific SNR. E.g. to mix with an SNR of 10dB when both features_a and features_b energies are 100, the features_b signal energy needs to be scaled by 0.1. Since different features (e.g. spectrogram, fbank, MFCC) require different combination of transformations (e.g. exp, log, sqrt, pow) to allow mixing of two signals, the exact place where to apply energy_scaling_factor_b to the signal is determined by the implementer.

Return type ndarray

Returns A mixed feature matrix.

static **compute_energy**(*features*)

Compute the total energy of a feature matrix. How the energy is computed depends on a particular type of features. It is expected that when implemented, compute_energy will never return zero.

Parameters **features** (ndarray) – A feature matrix.

Return type float

Returns A positive float value of the signal energy.

`__init__`(*config=None*)

property device: Union[str, torch.device]

Return type Union[str, device]

`extract_batch`(*samples*, *sampling_rate*)

Performs batch extraction. It is not guaranteed to be faster than `FeatureExtractor.extract()` – it depends on whether the implementation of a particular feature extractor supports accelerated batch computation.

Note: Unless overridden by child classes, it defaults to sequentially calling `FeatureExtractor.extract()` on the inputs.

Note: This method *should* support variable length inputs.

Return type Union[ndarray, Tensor, List[ndarray], List[Tensor]]

`extract_from_recording_and_store`(*recording*, *storage*, *offset=0*, *duration=None*, *channels=None*, *augment_fn=None*)

Extract the features from a `Recording` in a full pipeline:

- load audio from disk;
- optionally, perform audio augmentation;
- extract the features;
- save them to disk in a specified directory;
- return a `Features` object with a description of the extracted features and the source data used.

Parameters

- **recording** (*Recording*) – a `Recording` that specifies what's the input audio.
- **storage** (*FeaturesWriter*) – a `FeaturesWriter` object that will handle storing the feature matrices.
- **offset** (float) – an optional offset in seconds for where to start reading the recording.
- **duration** (Optional[float]) – an optional duration specifying how much audio to load from the recording.
- **channels** (Union[int, List[int], None]) – an optional int or list of ints, specifying the channels; by default, all channels will be used.
- **augment_fn** (Optional[Callable[[ndarray, int], ndarray]]) – an optional `WavAugmenter` instance to modify the waveform before feature extraction.

Return type *Features*

Returns a `Features` manifest item for the extracted feature matrix.

extract_from_samples_and_store(*samples*, *storage*, *sampling_rate*, *offset=0*, *channel=None*, *augment_fn=None*)

Extract the features from an array of audio samples in a full pipeline:

- optional audio augmentation;
- extract the features;
- save them to disk in a specified directory;
- return a `Features` object with a description of the extracted features.

Note, unlike in `extract_from_recording_and_store`, the returned `Features` object might not be suitable to store in a `FeatureSet`, as it does not reference any particular `Recording`. Instead, this method is useful when extracting features from cuts - especially `MixedCut` instances, which may be created from multiple recordings and channels.

Parameters

- **samples** (`ndarray`) – a numpy ndarray with the audio samples.
- **sampling_rate** (`int`) – integer sampling rate of `samples`.
- **storage** (*FeaturesWriter*) – a `FeaturesWriter` object that will handle storing the feature matrices.
- **offset** (`float`) – an offset in seconds for where to start reading the recording - when used for `Cut` feature extraction, must be equal to `Cut.start`.
- **channel** (`Optional[int]`) – an optional channel number to insert into `Features` manifest.
- **augment_fn** (`Optional[Callable[[ndarray, int], ndarray]]`) – an optional `WavAugmenter` instance to modify the waveform before feature extraction.

Return type *Features*

Returns a `Features` manifest item for the extracted feature matrix (it is not written to disk).

classmethod from_dict(*data*)

Return type *FeatureExtractor*

classmethod from_yaml(*path*)

Return type *FeatureExtractor*

to_dict()

Return type `Dict[str, Any]`

to_yaml(*path*)

9.3.6 Feature storage

class lhotse.features.io.**FeaturesWriter**

FeaturesWriter defines the interface of how to store numpy arrays in a particular storage backend. This backend could either be:

- separate files on a local filesystem;
- a single file with multiple arrays;
- cloud storage;
- etc.

Each class inheriting from FeaturesWriter must define:

- **the write() method, which defines the storing operation** (accepts a key used to place the value array in the storage);
- **the storage_path() property, which is either a common directory for the files,** the name of the file storing multiple arrays, name of the cloud bucket, etc.
- **the name() property that is unique to this particular storage mechanism -** it is stored in the features manifests (metadata) and used to automatically deduce the backend when loading the features.

Each *FeaturesWriter* can also be used as a context manager, as some implementations might need to free a resource after the writing is finalized. By default nothing happens in the context manager functions, and this can be modified by the inheriting subclasses.

Example:

```
>>> with MyWriter('some/path') as storage:
...     extractor.extract_from_recording_and_store(recording, storage)
```

The features loading must be defined separately in a class inheriting from *FeaturesReader*.

abstract property name: str

Return type str

abstract property storage_path: str

Return type str

abstract write(*key*, *value*)

Return type str

store_array(*key*, *value*, *frame_shift=None*, *temporal_dim=None*, *start=0*)

Store a numpy array in the underlying storage and return a manifest describing how to retrieve the data.

If the array contains a temporal dimension (e.g. it represents the frame-level features, alignment, posteriors, etc. of an utterance) then temporal_dim and frame_shift may be specified to enable downstream padding, truncating, and partial reads of the array.

Parameters

- **key** (str) – An ID that uniquely identifies the array.
- **value** (ndarray) – The array to be stored.
- **frame_shift** (Optional[float]) – Optional float, when the array has a temporal dimension it indicates how much time has passed between the starts of consecutive frames (expressed in seconds).

- **temporal_dim** (Optional[int]) – Optional int, when the array has a temporal dimension, it indicates which dim to interpret as temporal.
- **start** (float) – Float, when the array is temporal, it indicates what is the offset of the array w.r.t. the start of recording. Useful for reading subsets of an array when it represents something computed from long recordings. Ignored for non-temporal arrays.

Return type Union[Array, TemporalArray]

Returns A manifest of type Array or TemporalArray, depending on the input arguments.

class lhotse.features.io.FeaturesReader

FeaturesReader defines the interface of how to load numpy arrays from a particular storage backend. This backend could either be:

- separate files on a local filesystem;
- a single file with multiple arrays;
- cloud storage;
- etc.

Each class inheriting from FeaturesReader must define:

- **the read() method, which defines the loading operation** (accepts the key to locate the array in the storage and return it). The read method should support selecting only a subset of the feature matrix, with the bounds expressed as arguments left_offset_frames and right_offset_frames. It's up to the Reader implementation to load only the required part or trim it to that range only after loading. It is assumed that the time dimension is always the first one.
- **the name() property that is unique to this particular storage mechanism -** it is stored in the features manifests (metadata) and used to automatically deduce the backend when loading the features.

The features writing must be defined separately in a class inheriting from FeaturesWriter.

abstract property name: str

Return type str

abstract read(*key*, *left_offset_frames=0*, *right_offset_frames=None*)

Return type ndarray

lhotse.features.io.**available_storage_backends**()

Return type List[str]

lhotse.features.io.**register_reader**(*cls*)

Decorator used to add a new FeaturesReader to Lhotse's registry.

Example:

```
@register_reader
class MyFeatureReader(FeatureReader):
    ...
```

lhotse.features.io.**register_writer**(*cls*)

Decorator used to add a new FeaturesWriter to Lhotse's registry.

Example:

```
@register_writer
class MyFeatureWriter(FeatureWriter):
    ...
```

lhotse.features.io.**get_reader**(*name*)

Find a FeaturesReader sub-class that corresponds to the provided name and return its type.

Example:

reader_type = get_reader("lilcom_files") reader = reader_type("/storage/features/")

Return type Type[*FeaturesReader*]

lhotse.features.io.**get_writer**(*name*)

Find a FeaturesWriter sub-class that corresponds to the provided name and return its type.

Example:

writer_type = get_writer("lilcom_files") writer = writer_type("/storage/features/")

Return type Type[*FeaturesWriter*]

class lhotse.features.io.**LilcomFilesReader**(*storage_path*, **args*, ***kwargs*)

Reads Lilcom-compressed files from a directory on the local filesystem. storage_path corresponds to the directory path; storage_key for each utterance is the name of the file in that directory.

name = 'lilcom_files'

__init__(*storage_path*, **args*, ***kwargs*)

read(*key*, *left_offset_frames=0*, *right_offset_frames=None*)

Return type ndarray

class lhotse.features.io.**LilcomFilesWriter**(*storage_path*, *tick_power=- 5*, **args*, ***kwargs*)

Writes Lilcom-compressed files to a directory on the local filesystem. storage_path corresponds to the directory path; storage_key for each utterance is the name of the file in that directory.

name = 'lilcom_files'

__init__(*storage_path*, *tick_power=- 5*, **args*, ***kwargs*)

property storage_path: str

Return type str

write(*key*, *value*)

Return type str

store_array(*key*, *value*, *frame_shift=None*, *temporal_dim=None*, *start=0*)

Store a numpy array in the underlying storage and return a manifest describing how to retrieve the data.

If the array contains a temporal dimension (e.g. it represents the frame-level features, alignment, posteriors, etc. of an utterance) then temporal_dim and frame_shift may be specified to enable downstream padding, truncating, and partial reads of the array.

Parameters

- **key** (str) – An ID that uniquely identifies the array.

- **value** (ndarray) – The array to be stored.
- **frame_shift** (Optional[float]) – Optional float, when the array has a temporal dimension it indicates how much time has passed between the starts of consecutive frames (expressed in seconds).
- **temporal_dim** (Optional[int]) – Optional int, when the array has a temporal dimension, it indicates which dim to interpret as temporal.
- **start** (float) – Float, when the array is temporal, it indicates what is the offset of the array w.r.t. the start of recording. Useful for reading subsets of an array when it represents something computed from long recordings. Ignored for non-temporal arrays.

Return type Union[Array, TemporalArray]

Returns A manifest of type Array or TemporalArray, depending on the input arguments.

class lhotse.features.io.**NumpyFilesReader**(*storage_path*, **args*, ***kwargs*)

Reads non-compressed numpy arrays from files in a directory on the local filesystem. storage_path corresponds to the directory path; storage_key for each utterance is the name of the file in that directory.

name = 'numpy_files'

__init__(*storage_path*, **args*, ***kwargs*)

read(*key*, *left_offset_frames=0*, *right_offset_frames=None*)

Return type ndarray

class lhotse.features.io.**NumpyFilesWriter**(*storage_path*, **args*, ***kwargs*)

Writes non-compressed numpy arrays to files in a directory on the local filesystem. storage_path corresponds to the directory path; storage_key for each utterance is the name of the file in that directory.

name = 'numpy_files'

__init__(*storage_path*, **args*, ***kwargs*)

property storage_path: str

Return type str

write(*key*, *value*)

Return type str

store_array(*key*, *value*, *frame_shift=None*, *temporal_dim=None*, *start=0*)

Store a numpy array in the underlying storage and return a manifest describing how to retrieve the data.

If the array contains a temporal dimension (e.g. it represents the frame-level features, alignment, posteriors, etc. of an utterance) then temporal_dim and frame_shift may be specified to enable downstream padding, truncating, and partial reads of the array.

Parameters

- **key** (str) – An ID that uniquely identifies the array.
- **value** (ndarray) – The array to be stored.
- **frame_shift** (Optional[float]) – Optional float, when the array has a temporal dimension it indicates how much time has passed between the starts of consecutive frames (expressed in seconds).
- **temporal_dim** (Optional[int]) – Optional int, when the array has a temporal dimension, it indicates which dim to interpret as temporal.

- **start** (float) – Float, when the array is temporal, it indicates what is the offset of the array w.r.t. the start of recording. Useful for reading subsets of an array when it represents something computed from long recordings. Ignored for non-temporal arrays.

Return type Union[Array, TemporalArray]

Returns A manifest of type Array or TemporalArray, depending on the input arguments.

lhotse.features.io.**lookup_cache_or_open**(*storage_path*)

Helper internal function used in HDF5 readers. It opens the HDF files and keeps their handles open in a global program cache to avoid excessive amount of syscalls when the Reader class is instantiated and destroyed in a loop repeatedly (frequent use-case).

The file handles can be freed at any time by calling close_cached_file_handles().

lhotse.features.io.**lookup_cache_or_open_regular_file**(*storage_path*)

Helper internal function used in "fast" file readers. It opens regular files and keeps their handles open in a global program cache to avoid excessive amount of syscalls when the Reader class is instantiated and destroyed in a loop repeatedly (frequent use-case).

The file handles can be freed at any time by calling close_cached_file_handles().

lhotse.features.io.**lookup_chunk_size**(*h5_file_handle*)

Helper internal function to retrieve the chunk size from an HDF5 file. Helps avoid unnecessary repeated disk reads.

Return type int

lhotse.features.io.**close_cached_file_handles**()

Closes the cached file handles in lookup_cache_or_open (see its docs for more details).

Return type None

class lhotse.features.io.**NumpyHdf5Reader**(*storage_path*, **args*, ***kwargs*)

Reads non-compressed numpy arrays from a HDF5 file with a "flat" layout. Each array is stored as a separate HDF Dataset because their shapes (numbers of frames) may vary. storage_path corresponds to the HDF5 file path; storage_key for each utterance is the key corresponding to the array (i.e. HDF5 "Group" name).

name = 'numpy_hdf5'

__init__(*storage_path*, **args*, ***kwargs*)

read(*key*, *left_offset_frames=0*, *right_offset_frames=None*)

Return type ndarray

class lhotse.features.io.**NumpyHdf5Writer**(*storage_path*, *mode='w'*, **args*, ***kwargs*)

Writes non-compressed numpy arrays to a HDF5 file with a "flat" layout. Each array is stored as a separate HDF Dataset because their shapes (numbers of frames) may vary. storage_path corresponds to the HDF5 file path; storage_key for each utterance is the key corresponding to the array (i.e. HDF5 "Group" name).

Internally, this class opens the file lazily so that this object can be passed between processes without issues. This simplifies the parallel feature extraction code.

name = 'numpy_hdf5'

__init__(*storage_path*, *mode='w'*, **args*, ***kwargs*)

Parameters

- **storage_path** (Union[Path, str]) – Path under which we'll create the HDF5 file. We will add a .h5 suffix if it is not already in storage_path.

- **mode** (str) – Modes supported by h5py: w Create file, truncate if exists (default) w- or x Create file, fail if exists a Read/write if exists, create otherwise

property storage_path: str

Return type str

write(*key*, *value*)

Return type str

close()

Return type None

store_array(*key*, *value*, *frame_shift=None*, *temporal_dim=None*, *start=0*)

Store a numpy array in the underlying storage and return a manifest describing how to retrieve the data.

If the array contains a temporal dimension (e.g. it represents the frame-level features, alignment, posteriors, etc. of an utterance) then `temporal_dim` and `frame_shift` may be specified to enable downstream padding, truncating, and partial reads of the array.

Parameters

- **key** (str) – An ID that uniquely identifies the array.
- **value** (ndarray) – The array to be stored.
- **frame_shift** (Optional[float]) – Optional float, when the array has a temporal dimension it indicates how much time has passed between the starts of consecutive frames (expressed in seconds).
- **temporal_dim** (Optional[int]) – Optional int, when the array has a temporal dimension, it indicates which dim to interpret as temporal.
- **start** (float) – Float, when the array is temporal, it indicates what is the offset of the array w.r.t. the start of recording. Useful for reading subsets of an array when it represents something computed from long recordings. Ignored for non-temporal arrays.

Return type Union[Array, TemporalArray]

Returns A manifest of type `Array` or `TemporalArray`, depending on the input arguments.

class lhotse.features.io.**LilcomHdf5Reader**(*storage_path*, **args*, ***kwargs*)

Reads lilcom-compressed numpy arrays from a HDF5 file with a "flat" layout. Each array is stored as a separate HDF `Dataset` because their shapes (numbers of frames) may vary. `storage_path` corresponds to the HDF5 file path; `storage_key` for each utterance is the key corresponding to the array (i.e. HDF5 "Group" name).

name = 'lilcom_hdf5'

__init__(*storage_path*, **args*, ***kwargs*)

read(*key*, *left_offset_frames=0*, *right_offset_frames=None*)

Return type ndarray

class lhotse.features.io.**LilcomHdf5Writer**(*storage_path*, *tick_power=- 5*, *mode='w'*, **args*, ***kwargs*)

Writes lilcom-compressed numpy arrays to a HDF5 file with a "flat" layout. Each array is stored as a separate HDF `Dataset` because their shapes (numbers of frames) may vary. `storage_path` corresponds to the HDF5 file path; `storage_key` for each utterance is the key corresponding to the array (i.e. HDF5 "Group" name).

name = 'lilcom_hdf5'

`__init__`(*storage_path*, *tick_power=- 5*, *mode='w'*, **args*, ***kwargs*)

Parameters

- **storage_path** (Union[Path, str]) – Path under which we'll create the HDF5 file. We will add a .h5 suffix if it is not already in storage_path.
- **tick_power** (int) – Determines the lilcom compression accuracy; the input will be compressed to integer multiples of 2^tick_power.
- **mode** (str) – Modes supported by h5py: w Create file, truncate if exists (default) w- or x Create file, fail if exists a Read/write if exists, create otherwise

property storage_path: str

Return type str

`write`(*key*, *value*)

Return type str

`close`()

Return type None

`store_array`(*key*, *value*, *frame_shift=None*, *temporal_dim=None*, *start=0*)

Store a numpy array in the underlying storage and return a manifest describing how to retrieve the data.

If the array contains a temporal dimension (e.g. it represents the frame-level features, alignment, posteriors, etc. of an utterance) then temporal_dim and frame_shift may be specified to enable downstream padding, truncating, and partial reads of the array.

Parameters

- **key** (str) – An ID that uniquely identifies the array.
- **value** (ndarray) – The array to be stored.
- **frame_shift** (Optional[float]) – Optional float, when the array has a temporal dimension it indicates how much time has passed between the starts of consecutive frames (expressed in seconds).
- **temporal_dim** (Optional[int]) – Optional int, when the array has a temporal dimension, it indicates which dim to interpret as temporal.
- **start** (float) – Float, when the array is temporal, it indicates what is the offset of the array w.r.t. the start of recording. Useful for reading subsets of an array when it represents something computed from long recordings. Ignored for non-temporal arrays.

Return type Union[Array, TemporalArray]

Returns A manifest of type Array or TemporalArray, depending on the input arguments.

class lhotse.features.io.**ChunkedLilcomHdf5Reader**(*storage_path*, **args*, ***kwargs*)

Reads lilcom-compressed numpy arrays from a HDF5 file with chunked lilcom storage. Each feature matrix is stored in an array of chunks - binary data compressed with lilcom. Upon reading, we check how many chunks need to be retrieved to avoid excessive I/O.

storage_path corresponds to the HDF5 file path; storage_key for each utterance is the key corresponding to the array (i.e. HDF5 "Group" name).

name = 'chunked_lilcom_hdf5'

__init__(*storage_path*, **args*, ***kwargs*)

read(*key*, *left_offset_frames=0*, *right_offset_frames=None*)

Return type ndarray

class lhotse.features.io.**ChunkedLilcomHdf5Writer**(*storage_path*, *tick_power=- 5*, *chunk_size=100*, *mode='w'*, **args*, ***kwargs*)

Writes lilcom-compressed numpy arrays to a HDF5 file with chunked lilcom storage. Each feature matrix is stored in an array of chunks - binary data compressed with lilcom. Upon reading, we check how many chunks need to be retrieved to avoid excessive I/O.

`storage_path` corresponds to the HDF5 file path; `storage_key` for each utterance is the key corresponding to the array (i.e. HDF5 "Group" name).

name = 'chunked_lilcom_hdf5'

__init__(*storage_path*, *tick_power=- 5*, *chunk_size=100*, *mode='w'*, **args*, ***kwargs*)

Parameters

- **storage_path** (Union[Path, str]) – Path under which we'll create the HDF5 file. We will add a `.h5` suffix if it is not already in `storage_path`.
- **tick_power** (int) – Determines the lilcom compression accuracy; the input will be compressed to integer multiples of 2^tick_power.
- **chunk_size** (int) – How many frames to store per chunk. Too low a number will require many reads for long feature matrices, too high a number will require to read more redundant data.
- **mode** (str) – Modes supported by h5py: w Create file, truncate if exists (default) w- or x Create file, fail if exists a Read/write if exists, create otherwise

property storage_path: str

Return type str

write(*key*, *value*)

Return type str

close()

Return type None

store_array(*key*, *value*, *frame_shift=None*, *temporal_dim=None*, *start=0*)

Store a numpy array in the underlying storage and return a manifest describing how to retrieve the data.

If the array contains a temporal dimension (e.g. it represents the frame-level features, alignment, posteriors, etc. of an utterance) then `temporal_dim` and `frame_shift` may be specified to enable downstream padding, truncating, and partial reads of the array.

Parameters

- **key** (str) – An ID that uniquely identifies the array.
- **value** (ndarray) – The array to be stored.

- **frame_shift** (Optional[float]) – Optional float, when the array has a temporal dimension it indicates how much time has passed between the starts of consecutive frames (expressed in seconds).
- **temporal_dim** (Optional[int]) – Optional int, when the array has a temporal dimension, it indicates which dim to interpret as temporal.
- **start** (float) – Float, when the array is temporal, it indicates what is the offset of the array w.r.t. the start of recording. Useful for reading subsets of an array when it represents something computed from long recordings. Ignored for non-temporal arrays.

Return type Union[Array, TemporalArray]

Returns A manifest of type Array or TemporalArray, depending on the input arguments.

class lhotse.features.io.**LilcomChunkyReader**(*storage_path*, **args*, ***kwargs*)

Reads lilcom-compressed numpy arrays from a binary file with chunked lilcom storage. Each feature matrix is stored in an array of chunks - binary data compressed with lilcom. Upon reading, we check how many chunks need to be retrieved to avoid excessive I/O.

storage_path corresponds to the binary file path.

storage_key for each utterance is a comma separated list of offsets in the file. The first number is the offset for the whole array, and the following numbers are relative offsets for each chunk. These offsets are relative to the previous chunk start.

name = 'lilcom_chunky'

CHUNK_SIZE = 500

__init__(*storage_path*, **args*, ***kwargs*)

read(*key*, *left_offset_frames=0*, *right_offset_frames=None*)

Return type ndarray

class lhotse.features.io.**LilcomChunkyWriter**(*storage_path*, *tick_power=- 5*, *mode='wb'*, **args*, ***kwargs*)

Writes lilcom-compressed numpy arrays to a binary file with chunked lilcom storage. Each feature matrix is stored in an array of chunks - binary data compressed with lilcom. Upon reading, we check how many chunks need to be retrieved to avoid excessive I/O.

storage_path corresponds to the binary file path.

storage_key for each utterance is a comma separated list of offsets in the file. The first number is the offset for the whole array, and the following numbers are relative offsets for each chunk. These offsets are relative to the previous chunk start.

name = 'lilcom_chunky'

CHUNK_SIZE = 500

__init__(*storage_path*, *tick_power=- 5*, *mode='wb'*, **args*, ***kwargs*)

Parameters

- **storage_path** (Union[Path, str]) – Path under which we'll create the binary file.
- **tick_power** (int) – Determines the lilcom compression accuracy; the input will be compressed to integer multiples of 2^tick_power.
- **chunk_size** – How many frames to store per chunk. Too low a number will require many reads for long feature matrices, too high a number will require to read more redundant data.

- **mode** (str) – Modes, one of: "w" (write) or "a" (append); can be "wb" and "ab", "b" is implicit

property storage_path: str

Return type str

write(*key*, *value*)

Return type str

close()

Return type None

store_array(*key*, *value*, *frame_shift=None*, *temporal_dim=None*, *start=0*)

Store a numpy array in the underlying storage and return a manifest describing how to retrieve the data.

If the array contains a temporal dimension (e.g. it represents the frame-level features, alignment, posteriors, etc. of an utterance) then `temporal_dim` and `frame_shift` may be specified to enable downstream padding, truncating, and partial reads of the array.

Parameters

- **key** (str) – An ID that uniquely identifies the array.
- **value** (ndarray) – The array to be stored.
- **frame_shift** (Optional[float]) – Optional float, when the array has a temporal dimension it indicates how much time has passed between the starts of consecutive frames (expressed in seconds).
- **temporal_dim** (Optional[int]) – Optional int, when the array has a temporal dimension, it indicates which dim to interpret as temporal.
- **start** (float) – Float, when the array is temporal, it indicates what is the offset of the array w.r.t. the start of recording. Useful for reading subsets of an array when it represents something computed from long recordings. Ignored for non-temporal arrays.

Return type Union[Array, TemporalArray]

Returns A manifest of type `Array` or `TemporalArray`, depending on the input arguments.

class lhotse.features.io.**LilcomURLReader**(*storage_path*, **args*, ***kwargs*)

Downloads Lilcom-compressed files from a URL (S3, GCP, Azure, HTTP, etc.). `storage_path` corresponds to the root URL (e.g. "s3://my-data-bucket") `storage_key` will be concatenated to `storage_path` to form a full URL (e.g. "my-feature-file.llc")

> **Caution:** Requires `smart_open` to be installed (`pip install smart_open`).

name = 'lilcom_url'

__init__(*storage_path*, **args*, ***kwargs*)

read(*key*, *left_offset_frames=0*, *right_offset_frames=None*)

Return type ndarray

class lhotse.features.io.**LilcomURLWriter**(*storage_path*, *tick_power=- 5*, **args*, ***kwargs*)

Writes Lilcom-compressed files to a URL (S3, GCP, Azure, HTTP, etc.). `storage_path` corresponds to the root URL (e.g. "s3://my-data-bucket") `storage_key` will be concatenated to `storage_path` to form a full URL (e.g. "my-feature-file.llc")

> **Caution:** Requires `smart_open` to be installed (`pip install smart_open`).

name = 'lilcom_url'

__init__(*storage_path*, *tick_power=- 5*, **args*, ***kwargs*)

property storage_path: str

Return type `str`

write(*key*, *value*)

Return type `str`

store_array(*key*, *value*, *frame_shift=None*, *temporal_dim=None*, *start=0*)

Store a numpy array in the underlying storage and return a manifest describing how to retrieve the data.

If the array contains a temporal dimension (e.g. it represents the frame-level features, alignment, posteriors, etc. of an utterance) then `temporal_dim` and `frame_shift` may be specified to enable downstream padding, truncating, and partial reads of the array.

Parameters

- **key** (`str`) – An ID that uniquely identifies the array.
- **value** (`ndarray`) – The array to be stored.
- **frame_shift** (`Optional[float]`) – Optional float, when the array has a temporal dimension it indicates how much time has passed between the starts of consecutive frames (expressed in seconds).
- **temporal_dim** (`Optional[int]`) – Optional int, when the array has a temporal dimension, it indicates which dim to interpret as temporal.
- **start** (`float`) – Float, when the array is temporal, it indicates what is the offset of the array w.r.t. the start of recording. Useful for reading subsets of an array when it represents something computed from long recordings. Ignored for non-temporal arrays.

Return type `Union[Array, TemporalArray]`

Returns A manifest of type `Array` or `TemporalArray`, depending on the input arguments.

class lhotse.features.io.**KaldiReader**(*storage_path*, **args*, ***kwargs*)

Reads Kaldi's "feats.scp" file using kaldi_native_io. `storage_path` corresponds to the path to `feats.scp`. `storage_key` corresponds to the utterance-id in Kaldi.

> **Caution:** Requires `kaldi_native_io` to be installed (`pip install kaldi_native_io`).

name = 'kaldiio'

__init__(*storage_path*, **args*, ***kwargs*)

read(*key*, *left_offset_frames=0*, *right_offset_frames=None*)

Return type ndarray

class lhotse.features.io.**KaldiWriter**(*storage_path*, *compression_method=1*, **args*, ***kwargs*)

Write data to Kaldi's "feats.scp" and "feats.ark" files using kaldi_native_io. `storage_path` corresponds to a directory where we'll create "feats.scp" and "feats.ark" files. `storage_key` corresponds to the utterance-id in Kaldi.

The following `compression_method` values are supported by kaldi_native_io:

```
kAutomaticMethod = 1
kSpeechFeature = 2
kTwoByteAuto = 3
kTwoByteSignedInteger = 4
kOneByteAuto = 5
kOneByteUnsignedInteger = 6
kOneByteZeroOne = 7
```

Note: Setting compression_method works only with 2D arrays.

Example:

```
>>> data = np.random.randn(131, 80)
>>> with KaldiWriter('featdir') as w:
...     w.write('utt1', data)
>>> reader = KaldiReader('featdir/feats.scp')
>>> read_data = reader.read('utt1')
>>> np.testing.assert_equal(data, read_data)
```

Caution: Requires `kaldi_native_io` to be installed (`pip install kaldi_native_io`).

name = 'kaldiio'

__init__(*storage_path*, *compression_method=1*, **args*, ***kwargs*)

property storage_path: str

Return type str

write(*key*, *value*)

Return type str

close()

Return type None

store_array(*key*, *value*, *frame_shift=None*, *temporal_dim=None*, *start=0*)

Store a numpy array in the underlying storage and return a manifest describing how to retrieve the data.

If the array contains a temporal dimension (e.g. it represents the frame-level features, alignment, posteriors, etc. of an utterance) then `temporal_dim` and `frame_shift` may be specified to enable downstream padding, truncating, and partial reads of the array.

Parameters

- **key** (str) – An ID that uniquely identifies the array.
- **value** (ndarray) – The array to be stored.
- **frame_shift** (Optional[float]) – Optional float, when the array has a temporal dimension it indicates how much time has passed between the starts of consecutive frames (expressed in seconds).
- **temporal_dim** (Optional[int]) – Optional int, when the array has a temporal dimension, it indicates which dim to interpret as temporal.
- **start** (float) – Float, when the array is temporal, it indicates what is the offset of the array w.r.t. the start of recording. Useful for reading subsets of an array when it represents something computed from long recordings. Ignored for non-temporal arrays.

Return type Union[Array, TemporalArray]

Returns A manifest of type Array or TemporalArray, depending on the input arguments.

lhotse.features.io.**get_memory_writer**(*name*)

class lhotse.features.io.**MemoryLilcomReader**(**args*, ***kwargs*)

name = 'memory_lilcom'

__init__(**args*, ***kwargs*)

read(*raw_data*, *left_offset_frames=0*, *right_offset_frames=None*)

Return type ndarray

class lhotse.features.io.**MemoryLilcomWriter**(**args*, ***kwargs*)

name = 'memory_lilcom'

__init__(**args*, ***kwargs*)

property storage_path: None

Return type None

write(*key*, *value*)

Return type bytes

close()

Return type None

store_array(*key*, *value*, *frame_shift=None*, *temporal_dim=None*, *start=0*)

Store a numpy array in the underlying storage and return a manifest describing how to retrieve the data.

If the array contains a temporal dimension (e.g. it represents the frame-level features, alignment, posteriors, etc. of an utterance) then temporal_dim and frame_shift may be specified to enable downstream padding, truncating, and partial reads of the array.

Parameters

- **key** (str) – An ID that uniquely identifies the array.
- **value** (ndarray) – The array to be stored.

- **frame_shift** (Optional[float]) – Optional float, when the array has a temporal dimension it indicates how much time has passed between the starts of consecutive frames (expressed in seconds).
- **temporal_dim** (Optional[int]) – Optional int, when the array has a temporal dimension, it indicates which dim to interpret as temporal.
- **start** (float) – Float, when the array is temporal, it indicates what is the offset of the array w.r.t. the start of recording. Useful for reading subsets of an array when it represents something computed from long recordings. Ignored for non-temporal arrays.

Return type Union[Array, TemporalArray]

Returns A manifest of type Array or TemporalArray, depending on the input arguments.

class lhotse.features.io.**MemoryRawReader**(**args*, ***kwargs*)

name = 'memory_raw'

__init__(**args*, ***kwargs*)

read(*raw_data*, *left_offset_frames=0*, *right_offset_frames=None*)

Return type ndarray

class lhotse.features.io.**MemoryRawWriter**(**args*, ***kwargs*)

name = 'memory_raw'

__init__(**args*, ***kwargs*)

property storage_path: None

Return type None

write(*key*, *value*)

Return type bytes

close()

Return type None

store_array(*key*, *value*, *frame_shift=None*, *temporal_dim=None*, *start=0*)

Store a numpy array in the underlying storage and return a manifest describing how to retrieve the data.

If the array contains a temporal dimension (e.g. it represents the frame-level features, alignment, posteriors, etc. of an utterance) then temporal_dim and frame_shift may be specified to enable downstream padding, truncating, and partial reads of the array.

Parameters

- **key** (str) – An ID that uniquely identifies the array.
- **value** (ndarray) – The array to be stored.
- **frame_shift** (Optional[float]) – Optional float, when the array has a temporal dimension it indicates how much time has passed between the starts of consecutive frames (expressed in seconds).

- **temporal_dim** (Optional[int]) – Optional int, when the array has a temporal dimension, it indicates which dim to interpret as temporal.
- **start** (float) – Float, when the array is temporal, it indicates what is the offset of the array w.r.t. the start of recording. Useful for reading subsets of an array when it represents something computed from long recordings. Ignored for non-temporal arrays.

Return type Union[Array, TemporalArray]

Returns A manifest of type Array or TemporalArray, depending on the input arguments.

lhotse.features.io.**pairwise**(*iterable*)
s -> (s0,s1), (s1,s2), (s2, s3), …

9.3.7 Feature-domain mixing

class lhotse.features.mixer.**FeatureMixer**(*feature_extractor*, *base_feats*, *frame_shift*, *padding_value=-1000.0*, *reference_energy=None*)

Utility class to mix multiple feature matrices into a single one. It should be instantiated separately for each mixing session (i.e. each MixedCut will create a separate FeatureMixer to mix its tracks). It is initialized with a numpy array of features (typically float32) that represents the "reference" signal for the mix. Other signals can be mixed to it with different time offsets and SNRs using the add_to_mix method. The time offset is relative to the start of the reference signal (only positive values are supported). The SNR is relative to the energy of the signal used to initialize the FeatureMixer.

It relies on the FeatureExtractor to have defined mix and compute_energy methods, so that the FeatureMixer knows how to scale and add two feature matrices together.

__init__(*feature_extractor*, *base_feats*, *frame_shift*, *padding_value=- 1000.0*, *reference_energy=None*)
FeatureMixer's constructor.

Parameters

- **feature_extractor** (*FeatureExtractor*) – The FeatureExtractor instance that specifies how to mix the features.
- **base_feats** (ndarray) – The features used to initialize the FeatureMixer are a point of reference in terms of energy and offset for all features mixed into them.
- **frame_shift** (float) – Required to correctly compute offset and padding during the mix.
- **padding_value** (float) – The value used to pad the shorter features during the mix. This value is adequate only for log space features. For non-log space features, e.g. energies, use either 0 or a small positive value like 1e-5.
- **reference_energy** (Optional[float]) – Optionally pass a reference energy value to compute SNRs against. This might be required when base_feats correspond to padding energies.

property **num_features**

property **unmixed_feats: numpy.ndarray**
Return a numpy ndarray with the shape (num_tracks, num_frames, num_features), where each track's feature matrix is padded and scaled adequately to the offsets and SNR used in add_to_mix call.

Return type ndarray

property **mixed_feats: numpy.ndarray**
Return a numpy ndarray with the shape (num_frames, num_features) - a mono mixed feature matrix of the tracks supplied with add_to_mix calls.

Return type ndarray

add_to_mix(*feats*, *sampling_rate*, *snr=None*, *offset=0.0*)
Add feature matrix of a new track into the mix. :type feats: `ndarray` :param feats: A 2D feature matrix to be mixed in. :type sampling_rate: `int` :param sampling_rate: The sampling rate of `feats` :type snr: `Optional[float]` :param snr: Signal-to-noise ratio, assuming `feats` represents noise (positive SNR - lower `feats` energy, negative SNR - higher `feats` energy) :type offset: `float` :param offset: How many seconds to shift `feats` in time. For mixing, the signal will be padded before the start with low energy values.

9.4 Augmentation

9.5 Cuts

Data structures and tools used to create training/testing examples.

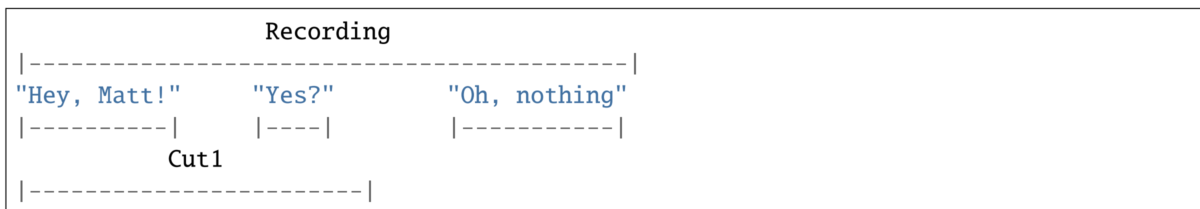
class lhotse.cut.**Cut**

> **Caution:** *Cut* is just an abstract class – the actual logic is implemented by its child classes (scroll down for references).

Cut is a base class for audio cuts. An "audio cut" is a subset of a *Recording* – it can also be thought of as a "view" or a pointer to a chunk of audio. It is not limited to audio data – cuts may also point to (sub-spans of) precomputed *Features*.

Cuts are different from *SupervisionSegment* in that they may be arbitrarily longer or shorter than supervisions; cuts may even contain multiple supervisions for creating contextual training data, and unsupervised regions that provide real or synthetic acoustic background context for the supervised segments.

The following example visualizes how a cut may represent a part of a single-channel recording with two utterances and some background noise in between:



This scenario can be represented in code, using *MonoCut*, as:

```
>>> from lhotse import Recording, SupervisionSegment, MonoCut
>>> rec = Recording(id='rec1', duration=10.0, sampling_rate=8000, num_samples=80000,
↪ sources=[...])
>>> sups = [
...     SupervisionSegment(id='sup1', recording_id='rec1', start=0, duration=3.37,␣
↪text='Hey, Matt!'),
...     SupervisionSegment(id='sup2', recording_id='rec1', start=4.5, duration=0.9,␣
↪text='Yes?'),
...     SupervisionSegment(id='sup3', recording_id='rec1', start=6.9, duration=2.9,␣
↪text='Oh, nothing'),
... ]
```

(continues on next page)

(continued from previous page)

```
>>> cut = MonoCut(id='rec1-cut1', start=0.0, duration=6.0, channel=0, recording=rec,
...     supervisions=[sups[0], sups[1]])
```

Note: All Cut classes assume that the *SupervisionSegment* time boundaries are relative to the beginning of the cut. E.g. if the underlying *Recording* starts at 0s (always true), the cut starts at 100s, and the SupervisionSegment inside the cut starts at 3s, it really did start at 103rd second of the recording. In some cases, the supervision might have a negative start, or a duration exceeding the duration of the cut; this means that the supervision in the recording extends beyond the cut.

Cut allows to check and read audio data or features data:

```
>>> assert cut.has_recording
>>> samples = cut.load_audio()
>>> if cut.has_features:
...     feats = cut.load_features()
```

It can be visualized, and listened to, inside Jupyter Notebooks:

```
>>> cut.plot_audio()
>>> cut.play_audio()
>>> cut.plot_features()
```

Cuts can be used with Lhotse's *FeatureExtractor* to compute features.

```
>>> from lhotse import Fbank
>>> feats = cut.compute_features(extractor=Fbank())
```

It is also possible to use a *FeaturesWriter* to store the features and attach their manifest to a copy of the cut:

```
>>> from lhotse import LilcomChunkyWriter
>>> with LilcomChunkyWriter('feats.lca') as storage:
...     cut_with_feats = cut.compute_and_store_features(
...         extractor=Fbank(),
...         storage=storage
...     )
```

Cuts have several methods that allow their manipulation, transformation, and mixing. Some examples (see the respective methods documentation for details):

```
>>> cut_2_to_4s = cut.truncate(offset=2, duration=2)
>>> cut_padded = cut.pad(duration=10.0)
>>> cut_extended = cut.extend_by(duration=5.0, direction='both')
>>> cut_mixed = cut.mix(other_cut, offset_other_by=5.0, snr=20)
>>> cut_append = cut.append(other_cut)
>>> cut_24k = cut.resample(24000)
>>> cut_sp = cut.perturb_speed(1.1)
>>> cut_vp = cut.perturb_volume(2.)
>>> cut_rvb = cut.reverb_rir(rir_recording)
```

Note: All cut transformations are performed lazily, on-the-fly, upon calling `load_audio` or `load_features`.

The stored waveforms and features are untouched.

Caution: Operations on cuts are not mutating – they return modified copies of *Cut* objects, leaving the original object unmodified.

A *Cut* that contains multiple segments (SupervisionSegment) can be decayed into smaller cuts that correspond directly to supervisions:

```
>>> smaller_cuts = cut.trim_to_supervisions()
```

Cuts can be detached from parts of their metadata:

```
>>> cut_no_feat = cut.drop_features()
>>> cut_no_rec = cut.drop_recording()
>>> cut_no_sup = cut.drop_supervisions()
```

Finally, cuts provide convenience methods to compute feature frame and audio sample masks for supervised regions:

```
>>> sup_frames = cut.supervisions_feature_mask()
>>> sup_samples = cut.supervisions_audio_mask()
```

See also:

- *lhotse.cut.MonoCut*
- *lhotse.cut.MixedCut*
- *lhotse.cut.CutSet*

id: str

start: float

duration: float

sampling_rate: int

supervisions: List[*lhotse.supervision.SupervisionSegment*]

num_samples: Optional[int]

num_frames: Optional[int]

num_features: Optional[int]

frame_shift: Optional[float]

features_type: Optional[str]

has_recording: bool

has_features: bool

from_dict: Callable[[Dict], *lhotse.cut.Cut*]

load_audio: Callable[[], numpy.ndarray]

load_features: Callable[[], numpy.ndarray]

compute_and_store_features: Callable

drop_features: Callable

drop_recording: Callable

drop_supervisions: Callable

truncate: Callable

pad: Callable

extend_by: Callable

resample: Callable

perturb_speed: Callable

perturb_tempo: Callable

perturb_volume: Callable

reverb_rir: Callable

map_supervisions: Callable

merge_supervisions: Callable

filter_supervisions: Callable

fill_supervision: Callable

with_features_path_prefix: Callable

with_recording_path_prefix: Callable

property end: float

Return type `float`

to_dict()

Return type `dict`

property trimmed_supervisions: List[*lhotse.supervision.SupervisionSegment*]

Return the supervisions in this Cut that have modified time boundaries so as not to exceed the Cut's start or end.

Note that when `cut.supervisions` is called, the supervisions may have negative `start` values that indicate the supervision actually begins before the cut, or `end` values that exceed the Cut's duration (it means the supervision continued in the original recording after the Cut's ending).

> **Caution:** For some tasks such as speech recognition (ASR), trimmed supervisions could result in corrupted training data. This is because a part of the transcript might actually reside outside of the cut.

Return type `List[`*SupervisionSegment*`]`

split(*timestamp*)

Split a cut into two cuts at `timestamp`, which is measured from the start of the cut. For example, a [0s - 10s] cut split at 4s yields:

- left cut [0s - 4s]
- right cut [4s - 10s]

Return type Tuple[*Cut*, *Cut*]

mix(*other*, *offset_other_by=0.0*, *allow_padding=False*, *snr=None*, *preserve_id=None*)
Refer to **:function:`~lhotse.cut.mix`** documentation.

Return type *MixedCut*

append(*other*, *snr=None*, *preserve_id=None*)
Append the `other` Cut after the current Cut. Conceptually the same as `mix` but with an offset matching the current cuts length. Optionally scale down (positive SNR) or scale up (negative SNR) the `other` cut. Returns a MixedCut, which only keeps the information about the mix; actual mixing is performed during the call to `load_features`.

Parameters **preserve_id** (Optional[str]) – optional string ("left", "right"). When specified, append will preserve the cut ID of the left- or right-hand side argument. Otherwise, a new random ID is generated.

Return type *MixedCut*

compute_features(*extractor*, *augment_fn=None*)
Compute the features from this cut. This cut has to be able to load audio.

Parameters

- **extractor** (*FeatureExtractor*) – a `FeatureExtractor` instance used to compute the features.
- **augment_fn** (Optional[Callable[[ndarray, int], ndarray]]) – optional `WavAugmenter` instance for audio augmentation.

Return type ndarray

Returns a numpy ndarray with the computed features.

plot_audio()
Display a plot of the waveform. Requires matplotlib to be installed.

play_audio()
Display a Jupyter widget that allows to listen to the waveform. Works only in Jupyter notebook/lab or similar (e.g. Colab).

plot_features()
Display the feature matrix as an image. Requires matplotlib to be installed.

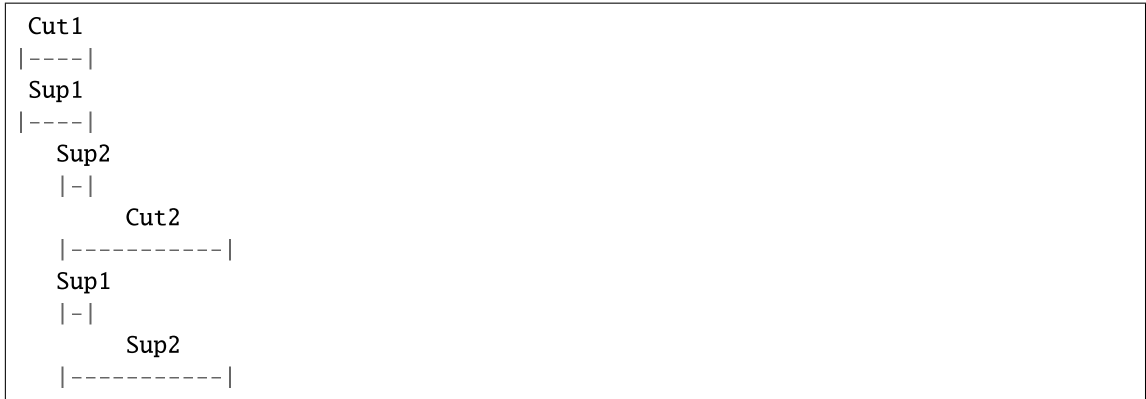
plot_alignment(*alignment_type='word'*)
Display the alignment on top of a spectrogram. Requires matplotlib to be installed.

trim_to_supervisions(*keep_overlapping=True*, *min_duration=None*, *context_direction='center'*)
Splits the current *Cut* into as many cuts as there are supervisions (*SupervisionSegment*). These cuts have identical start times and durations as the supervisions. When there are overlapping supervisions, they can be kept or discarded via `keep_overlapping` flag.

For example, the following cut:

```
        Cut
|-----------------|
 Sup1
|----|  Sup2
   |-----------|
```

is transformed into two cuts:



```
 Cut1
|----|
 Sup1
|----|
   Sup2
   |-|
        Cut2
   |-----------|
   Sup1
   |-|
        Sup2
   |-----------|
```

Parameters

- **keep_overlapping** (`bool`) – when `False`, it will discard parts of other supervisions that overlap with the main supervision. In the illustration above, it would discard `Sup2` in `Cut1` and `Sup1` in `Cut2`. In this mode, we guarantee that there will always be exactly one supervision per cut.
- **min_duration** (`Optional[float]`) – An optional duration in seconds; specifying this argument will extend the cuts that would have been shorter than `min_duration` with actual acoustic context in the recording/features. If there are supervisions present in the context, they are kept when `keep_overlapping` is true. If there is not enough context, the returned cut will be shorter than `min_duration`. If the supervision segment is longer than `min_duration`, the return cut will be longer.
- **context_direction** (`Literal`['center', 'left', 'right', 'random']) – Which direction should the cut be expanded towards to include context. The value of "center" implies equal expansion to left and right; random uniformly samples a value between "left" and "right".

Return type `List[Cut]`

Returns a list of cuts.

cut_into_windows(*duration*, *hop=None*, *keep_excessive_supervisions=True*)

Return a list of shorter cuts, made by traversing this cut in windows of `duration` seconds by `hop` seconds.

The last window might have a shorter duration if there was not enough audio, so you might want to use either filter or pad the results.

Parameters

- **duration** (`float`) – Desired duration of the new cuts in seconds.
- **hop** (`Optional[float]`) – Shift between the windows in the new cuts in seconds.
- **keep_excessive_supervisions** (`bool`) – bool. When a cut is truncated in the middle of a supervision segment, should the supervision be kept.

Return type `List[Cut]`

Returns a list of cuts made from shorter duration windows.

index_supervisions(*index_mixed_tracks=False*, *keep_ids=None*)

Create a two-level index of supervision segments. It is a mapping from a Cut's ID to an interval tree that contains the supervisions of that Cut.

The interval tree can be efficiently queried for overlapping and/or enveloping segments. It helps speed up some operations on Cuts of very long recordings (1h+) that contain many supervisions.

Parameters

- **index_mixed_tracks** (bool) – Should the tracks of MixedCut's be indexed as additional, separate entries.
- **keep_ids** (Optional[Set[str]]) – If specified, we will only index the supervisions with the specified IDs.

Return type Dict[str, IntervalTree]

Returns a mapping from Cut ID to an interval tree of SupervisionSegments.

compute_and_store_recording(*storage_path*, *augment_fn=None*)
Store this cut's waveform as audio recording to disk.

Parameters

- **storage_path** (Union[Path, str]) – The path to location where we will store the audio recordings.
- **augment_fn** (Optional[Callable[[ndarray, int], ndarray]]) – an optional callable used for audio augmentation. Be careful with the types of augmentations used: if they modify the start/end/duration times of the cut and its supervisions, you will end up with incorrect supervision information when using this API. E.g. for speed perturbation, use CutSet.perturb_speed() instead.

Return type *MonoCut*

Returns a new MonoCut instance.

save_audio(*storage_path*, *augment_fn=None*)
Store this cut's waveform as audio recording to disk.

Parameters

- **storage_path** (Union[Path, str]) – The path to location where we will store the audio recordings.
- **augment_fn** (Optional[Callable[[ndarray, int], ndarray]]) – an optional callable used for audio augmentation. Be careful with the types of augmentations used: if they modify the start/end/duration times of the cut and its supervisions, you will end up with incorrect supervision information when using this API. E.g. for speed perturbation, use CutSet.perturb_speed() instead.

Return type *MonoCut*

Returns a new MonoCut instance.

speakers_feature_mask(*min_speaker_dim=None*, *speaker_to_idx_map=None*, *use_alignment_if_exists=None*)
Return a matrix of per-speaker activity in a cut. The matrix shape is (num_speakers, num_frames), and its values are 0 for nonspeech **frames** and 1 for speech **frames** for each respective speaker.

This is somewhat inspired by the TS-VAD setup: https://arxiv.org/abs/2005.07272

Parameters

- **min_speaker_dim** (Optional[int]) – optional int, when specified it will enforce that the matrix shape is at least that value (useful for datasets like CHiME 6 where the number of speakers is always 4, but some cuts might have less speakers than that).
- **speaker_to_idx_map** (Optional[Dict[str, int]]) – optional dict mapping speaker names (strings) to their global indices (ints). Useful when you want to preserve the order of the speakers (e.g. speaker XYZ is always mapped to index 2)

- **use_alignment_if_exists** (Optional[str]) – optional str, key for alignment type to use for generating the mask. If not exists, fall back on supervision time spans.

Return type ndarray

speakers_audio_mask(*min_speaker_dim=None*, *speaker_to_idx_map=None*, *use_alignment_if_exists=None*)

Return a matrix of per-speaker activity in a cut. The matrix shape is (num_speakers, num_samples), and its values are 0 for nonspeech **samples** and 1 for speech **samples** for each respective speaker.

This is somewhat inspired by the TS-VAD setup: https://arxiv.org/abs/2005.07272

Parameters

- **min_speaker_dim** (Optional[int]) – optional int, when specified it will enforce that the matrix shape is at least that value (useful for datasets like CHiME 6 where the number of speakers is always 4, but some cuts might have less speakers than that).
- **speaker_to_idx_map** (Optional[Dict[str, int]]) – optional dict mapping speaker names (strings) to their global indices (ints). Useful when you want to preserve the order of the speakers (e.g. speaker XYZ is always mapped to index 2)
- **use_alignment_if_exists** (Optional[str]) – optional str, key for alignment type to use for generating the mask. If not exists, fall back on supervision time spans.

Return type ndarray

supervisions_feature_mask(*use_alignment_if_exists=None*)

Return a 1D numpy array with value 1 for **frames** covered by at least one supervision, and 0 for **frames** not covered by any supervision.

Parameters use_alignment_if_exists (Optional[str]) – optional str, key for alignment type to use for generating the mask. If not exists, fall back on supervision time spans.

Return type ndarray

supervisions_audio_mask(*use_alignment_if_exists=None*)

Return a 1D numpy array with value 1 for **samples** covered by at least one supervision, and 0 for **samples** not covered by any supervision.

Parameters use_alignment_if_exists (Optional[str]) – optional str, key for alignment type to use for generating the mask. If not exists, fall back on supervision time spans.

Return type ndarray

with_id(*id_*)

Return a copy of the Cut with a new ID.

Return type *Cut*

class lhotse.cut.**MonoCut**(*id*, *start*, *duration*, *channel*, *supervisions=<factory>*, *features=None*, *recording=None*, *custom=None*)

MonoCut is a *Cut* of a single channel of a *Recording*. In addition to Cut, it has a specified channel attribute. This is the most commonly used type of cut.

Please refer to the documentation of *Cut* to learn more about using cuts.

See also:

- *lhotse.cut.Cut*
- *lhotse.cut.MixedCut*
- *lhotse.cut.CutSet*

id: str

start: float

duration: float

channel: int

supervisions: List[*lhotse.supervision.SupervisionSegment*]

features: Optional[*lhotse.features.base.Features*] = None

recording: Optional[*lhotse.audio.Recording*] = None

custom: Optional[Dict[str, Any]] = None

load_custom(*name*)

Load custom data as numpy array. The custom data is expected to have been stored in cuts `custom` field as an `Array` or `TemporalArray` manifest.

Note: It works with Array manifests stored via attribute assignments, e.g.: `cut.my_custom_data = Array(...)`.

Parameters **name** (`str`) – name of the custom attribute.

Return type `ndarray`

Returns a numpy array with the data.

property recording_id: str

Return type `str`

property has_features: bool

Return type `bool`

property has_recording: bool

Return type `bool`

property frame_shift: Optional[float]

Return type `Optional[float]`

property num_frames: Optional[int]

Return type `Optional[int]`

property num_samples: Optional[int]

Return type `Optional[int]`

property num_features: Optional[int]

Return type `Optional[int]`

property features_type: Optional[str]

Return type `Optional[str]`

property sampling_rate: int

Return type `int`

load_features()

Load the features from the underlying storage and cut them to the relevant [begin, duration] region of the current MonoCut.

Return type Optional[ndarray]

load_audio()

Load the audio by locating the appropriate recording in the supplied RecordingSet. The audio is trimmed to the [begin, end] range specified by the MonoCut.

Return type Optional[ndarray]

Returns a numpy ndarray with audio samples, with shape (1 <channel>, N <samples>)

move_to_memory(*audio_format='flac'*, *load_audio=True*, *load_features=True*, *load_custom=True*)

Load data (audio, features, or custom arrays) into memory and attach them to a copy of the manifest. This is useful when you want to store cuts together with the actual data in some binary format that enables sequential data reads.

Audio is encoded with `audio_format` (compatible with `torchaudio.save`), floating point features are encoded with lilcom, and other arrays are pickled.

Return type *MonoCut*

drop_features()

Return a copy of the current *MonoCut*, detached from `features`.

Return type *MonoCut*

drop_recording()

Return a copy of the current *MonoCut*, detached from `recording`.

Return type *MonoCut*

drop_supervisions()

Return a copy of the current *MonoCut*, detached from `supervisions`.

Return type *MonoCut*

fill_supervision(*add_empty=True*, *shrink_ok=False*)

Fills the whole duration of a cut with a supervision segment.

If the cut has one supervision, its start is set to 0 and duration is set to `cut.duration`. Note: this may either expand a supervision that was shorter than a cut, or shrink a supervision that exceeds the cut.

If there are no supervisions, we will add an empty one when `add_empty==True`, otherwise we won't change anything.

If there are two or more supervisions, we will raise an exception.

Parameters

- **add_empty** (bool) – should we add an empty supervision with identical time bounds as the cut.
- **shrink_ok** (bool) – should we raise an error if a supervision would be shrank as a result of calling this method.

Return type *MonoCut*

compute_and_store_features(*extractor*, *storage*, *augment_fn=None*, **args*, ***kwargs*)

Compute the features from this cut, store them on disk, and attach a feature manifest to this cut. This cut has to be able to load audio.

Parameters

- **extractor** (*FeatureExtractor*) – a FeatureExtractor instance used to compute the features.
- **storage** (*FeaturesWriter*) – a FeaturesWriter instance used to write the features to a storage.
- **augment_fn** (Optional[Callable[[ndarray, int], ndarray]]) – an optional callable used for audio augmentation.

Return type *Cut*

Returns a new MonoCut instance with a Features manifest attached to it.

truncate(*, *offset=0.0*, *duration=None*, *keep_excessive_supervisions=True*, *preserve_id=False*, *_supervisions_index=None*)

Returns a new MonoCut that is a sub-region of the current MonoCut.

Note that no operation is done on the actual features or recording - it's only during the call to *MonoCut.load_features()* / *MonoCut.load_audio()* when the actual changes happen (a subset of features/audio is loaded).

Hint: To extend a cut by a fixed duration, use the *MonoCut.extend_by()* method.

Parameters

- **offset** (float) – float (seconds), controls the start of the new cut relative to the current MonoCut's start. E.g., if the current MonoCut starts at 10.0, and offset is 2.0, the new start is 12.0.
- **duration** (Optional[float]) – optional float (seconds), controls the duration of the resulting MonoCut. By default, the duration is (end of the cut before truncation) - (offset).
- **keep_excessive_supervisions** (bool) – bool. Since trimming may happen inside a SupervisionSegment, the caller has an option to either keep or discard such supervisions.
- **preserve_id** (bool) – bool. Should the truncated cut keep the same ID or get a new, random one.
- **_supervisions_index** (Optional[Dict[str, IntervalTree]]) – an IntervalTree; when passed, allows to speed up processing of Cuts with a very large number of supervisions. Intended as an internal parameter.

Return type *MonoCut*

Returns a new MonoCut instance. If the current MonoCut is shorter than the duration, return None.

extend_by(*, *duration*, *direction='both'*, *preserve_id=False*)

Returns a new MonoCut that is an extended region of the current MonoCut by extending the cut by a fixed duration in the specified direction.

Note that no operation is done on the actual features or recording - it's only during the call to *MonoCut.load_features()* / *MonoCut.load_audio()* when the actual changes happen (an extended version of features/audio is loaded).

Hint: This method extends a cut by a given duration, either to the left or to the right (or both), using the "real" content of the recording that the cut is part of. For example, a MonoCut spanning the region from 2s to 5s in a recording, when extended by 2s to the right, will now span the region from 2s to 7s in the same recording (provided the recording length exceeds 7s). If the recording is shorter, the cut will only

be extended up to the duration of the recording. To "expand" a cut by padding, use `MonoCut.pad()`. To "truncate" a cut, use `MonoCut.truncate()`.

Hint: If *direction* is "both", the resulting cut will be extended by the specified duration in both directions. This is different from the usage in `MonoCut.pad()` where a padding equal to 0.5*duration is added to both sides.

Parameters

- **duration** (float) – float (seconds), specifies the duration by which the cut should be extended.
- **direction** (str) – string, 'left', 'right' or 'both'. Determines whether to extend on the left, right, or both sides. If 'both', extend on both sides by the duration specified in *duration*.
- **preserve_id** (bool) – bool. Should the extended cut keep the same ID or get a new, random one.

Return type `MonoCut`

Returns a new MonoCut instance.

pad(*duration=None*, *num_frames=None*, *num_samples=None*, *pad_feat_value=- 23.025850929940457*, *direction='right'*, *preserve_id=False*, *pad_value_dict=None*)

Return a new MixedCut, padded with zeros in the recording, and `pad_feat_value` in each feature bin.

The user can choose to pad either to a specific *duration*; a specific number of frames *max_frames*; or a specific number of samples *num_samples*. The three arguments are mutually exclusive.

Parameters

- **duration** (Optional[float]) – The cut's minimal duration after padding.
- **num_frames** (Optional[int]) – The cut's total number of frames after padding.
- **num_samples** (Optional[int]) – The cut's total number of samples after padding.
- **pad_feat_value** (float) – A float value that's used for padding the features. By default we assume a log-energy floor of approx. -23 (1e-10 after exp).
- **direction** (str) – string, 'left', 'right' or 'both'. Determines whether the padding is added before or after the cut.
- **preserve_id** (bool) – When `True`, preserves the cut ID before padding. Otherwise, a new random ID is generated for the padded cut (default).
- **pad_value_dict** (Optional[Dict[str, Union[int, float]]]) – Optional dict that specifies what value should be used for padding arrays in custom attributes.

Return type `Cut`

Returns a padded MixedCut if duration is greater than this cut's duration, otherwise `self`.

resample(*sampling_rate*, *affix_id=False*)

Return a new `MonoCut` that will lazily resample the audio while reading it. This operation will drop the feature manifest, if attached. It does not affect the supervision.

Parameters

- **sampling_rate** (int) – The new sampling rate.

- **affix_id** (bool) – Should we modify the ID (useful if both versions of the same cut are going to be present in a single manifest).

Return type *MonoCut*

Returns a modified copy of the current MonoCut.

perturb_speed(*factor*, *affix_id=True*)
Return a new MonoCut that will lazily perturb the speed while loading audio. The num_samples, start and duration fields are updated to reflect the shrinking/extending effect of speed. We are also updating the time markers of the underlying Recording and the supervisions.

Parameters

- **factor** (float) – The speed will be adjusted this many times (e.g. factor=1.1 means 1.1x faster).
- **affix_id** (bool) – When true, we will modify the MonoCut.id field by affixing it with "_sp{factor}".

Return type *MonoCut*

Returns a modified copy of the current MonoCut.

perturb_tempo(*factor*, *affix_id=True*)
Return a new MonoCut that will lazily perturb the tempo while loading audio.

Compared to speed perturbation, tempo preserves pitch. The num_samples, start and duration fields are updated to reflect the shrinking/extending effect of speed. We are also updating the time markers of the underlying Recording and the supervisions.

Parameters

- **factor** (float) – The tempo will be adjusted this many times (e.g. factor=1.1 means 1.1x faster).
- **affix_id** (bool) – When true, we will modify the MonoCut.id field by affixing it with "_tp{factor}".

Return type *MonoCut*

Returns a modified copy of the current MonoCut.

perturb_volume(*factor*, *affix_id=True*)
Return a new MonoCut that will lazily perturb the volume while loading audio.

Parameters

- **factor** (float) – The volume will be adjusted this many times (e.g. factor=1.1 means 1.1x louder).
- **affix_id** (bool) – When true, we will modify the MonoCut.id field by affixing it with "_vp{factor}".

Return type *MonoCut*

Returns a modified copy of the current MonoCut.

reverb_rir(*rir_recording*, *normalize_output=True*, *early_only=False*, *affix_id=True*, *rir_channels=[0]*)
Return a new MonoCut that will convolve the audio with the provided impulse response. If the *rir_recording* is multi-channel, the *rir_channels* argument determines which channels will be used. By default, we use the first channel and return a MonoCut.

Parameters

- **rir_recording** (*Recording*) – The impulse response to use for convolving.

- **normalize_output** (bool) – When true, output will be normalized to have energy as input.
- **early_only** (bool) – When true, only the early reflections (first 50 ms) will be used.
- **affix_id** (bool) – When true, we will modify the MonoCut.id field by affixing it with "_rvb".
- **rir_channels** (List[int]) – The channels of the impulse response to use. First channel is used by default. If multiple channels are specified, this will produce a MixedCut instead of a MonoCut.

Return type Union[*MonoCut*, *MixedCut*]

Returns a modified copy of the current MonoCut.

map_supervisions(*transform_fn*)

Return a copy of the cut that has its supervisions transformed by transform_fn.

Parameters **transform_fn** (Callable[[*SupervisionSegment*], *SupervisionSegment*]) – a function that modifies a supervision as an argument.

Return type *MonoCut*

Returns a modified MonoCut.

filter_supervisions(*predicate*)

Return a copy of the cut that only has supervisions accepted by predicate.

Example:

```
>>> cut = cut.filter_supervisions(lambda s: s.id in supervision_ids)
>>> cut = cut.filter_supervisions(lambda s: s.duration < 5.0)
>>> cut = cut.filter_supervisions(lambda s: s.text is not None)
```

Parameters **predicate** (Callable[[*SupervisionSegment*], bool]) – A callable that accepts *SupervisionSegment* and returns bool

Return type *MonoCut*

Returns a modified MonoCut

merge_supervisions(*custom_merge_fn=None*)

Return a copy of the cut that has all of its supervisions merged into a single segment.

The new start is the start of the earliest superivion, and the new duration is a minimum spanning duration for all the supervisions.

The text fields are concatenated with a whitespace, and all other string fields (including IDs) are prefixed with "cat#" and concatenated with a hash symbol "#". This is also applied to custom fields. Fields with a None value are omitted.

Parameters **custom_merge_fn** (Optional[Callable[[str, Iterable[Any]], Any]]) – a function that will be called to merge custom fields values. We expect custom_merge_fn to handle all possible custom keys. When not provided, we will treat all custom values as strings. It will be called roughly like: custom_merge_fn(custom_key, [s.custom[custom_key] for s in sups])

Return type *MonoCut*

static from_dict(*data*)

Return type *MonoCut*

with_features_path_prefix(*path*)

Return type *MonoCut*

with_recording_path_prefix(*path*)

Return type *MonoCut*

__init__(*id*, *start*, *duration*, *channel*, *supervisions=<factory>*, *features=None*, *recording=None*, *custom=None*)

append(*other*, *snr=None*, *preserve_id=None*)

Append the `other` Cut after the current Cut. Conceptually the same as `mix` but with an offset matching the current cuts length. Optionally scale down (positive SNR) or scale up (negative SNR) the `other` cut. Returns a MixedCut, which only keeps the information about the mix; actual mixing is performed during the call to `load_features`.

Parameters **preserve_id** (`Optional[str]`) – optional string ("left", "right"). When specified, append will preserve the cut ID of the left- or right-hand side argument. Otherwise, a new random ID is generated.

Return type *MixedCut*

compute_and_store_recording(*storage_path*, *augment_fn=None*)

Store this cut's waveform as audio recording to disk.

Parameters

- **storage_path** (`Union[Path, str]`) – The path to location where we will store the audio recordings.
- **augment_fn** (`Optional[Callable[[ndarray, int], ndarray]]`) – an optional callable used for audio augmentation. Be careful with the types of augmentations used: if they modify the start/end/duration times of the cut and its supervisions, you will end up with incorrect supervision information when using this API. E.g. for speed perturbation, use `CutSet.perturb_speed()` instead.

Return type *MonoCut*

Returns a new MonoCut instance.

compute_features(*extractor*, *augment_fn=None*)

Compute the features from this cut. This cut has to be able to load audio.

Parameters

- **extractor** (*FeatureExtractor*) – a `FeatureExtractor` instance used to compute the features.
- **augment_fn** (`Optional[Callable[[ndarray, int], ndarray]]`) – optional `WavAugmenter` instance for audio augmentation.

Return type `ndarray`

Returns a numpy ndarray with the computed features.

cut_into_windows(*duration*, *hop=None*, *keep_excessive_supervisions=True*)

Return a list of shorter cuts, made by traversing this cut in windows of `duration` seconds by `hop` seconds.

The last window might have a shorter duration if there was not enough audio, so you might want to use either filter or pad the results.

Parameters

- **duration** (float) – Desired duration of the new cuts in seconds.
- **hop** (Optional[float]) – Shift between the windows in the new cuts in seconds.
- **keep_excessive_supervisions** (bool) – bool. When a cut is truncated in the middle of a supervision segment, should the supervision be kept.

Return type List[*Cut*]

Returns a list of cuts made from shorter duration windows.

property end: float

Return type float

index_supervisions(*index_mixed_tracks=False*, *keep_ids=None*)

Create a two-level index of supervision segments. It is a mapping from a Cut's ID to an interval tree that contains the supervisions of that Cut.

The interval tree can be efficiently queried for overlapping and/or enveloping segments. It helps speed up some operations on Cuts of very long recordings (1h+) that contain many supervisions.

Parameters

- **index_mixed_tracks** (bool) – Should the tracks of MixedCut's be indexed as additional, separate entries.
- **keep_ids** (Optional[Set[str]]) – If specified, we will only index the supervisions with the specified IDs.

Return type Dict[str, IntervalTree]

Returns a mapping from Cut ID to an interval tree of SupervisionSegments.

mix(*other*, *offset_other_by=0.0*, *allow_padding=False*, *snr=None*, *preserve_id=None*)

Refer to **:function:`~lhotse.cut.mix`** documentation.

Return type *MixedCut*

play_audio()

Display a Jupyter widget that allows to listen to the waveform. Works only in Jupyter notebook/lab or similar (e.g. Colab).

plot_alignment(*alignment_type='word'*)

Display the alignment on top of a spectrogram. Requires matplotlib to be installed.

plot_audio()

Display a plot of the waveform. Requires matplotlib to be installed.

plot_features()

Display the feature matrix as an image. Requires matplotlib to be installed.

save_audio(*storage_path*, *augment_fn=None*)

Store this cut's waveform as audio recording to disk.

Parameters

- **storage_path** (Union[Path, str]) – The path to location where we will store the audio recordings.
- **augment_fn** (Optional[Callable[[ndarray, int], ndarray]]) – an optional callable used for audio augmentation. Be careful with the types of augmentations used: if they modify the start/end/duration times of the cut and its supervisions, you will end up with

incorrect supervision information when using this API. E.g. for speed perturbation, use `CutSet.perturb_speed()` instead.

Return type *MonoCut*

Returns a new MonoCut instance.

speakers_audio_mask(*min_speaker_dim=None*, *speaker_to_idx_map=None*, *use_alignment_if_exists=None*)

Return a matrix of per-speaker activity in a cut. The matrix shape is (num_speakers, num_samples), and its values are 0 for nonspeech **samples** and 1 for speech **samples** for each respective speaker.

This is somewhat inspired by the TS-VAD setup: https://arxiv.org/abs/2005.07272

Parameters

- **min_speaker_dim** (`Optional[int]`) – optional int, when specified it will enforce that the matrix shape is at least that value (useful for datasets like CHiME 6 where the number of speakers is always 4, but some cuts might have less speakers than that).
- **speaker_to_idx_map** (`Optional[Dict[str, int]]`) – optional dict mapping speaker names (strings) to their global indices (ints). Useful when you want to preserve the order of the speakers (e.g. speaker XYZ is always mapped to index 2)
- **use_alignment_if_exists** (`Optional[str]`) – optional str, key for alignment type to use for generating the mask. If not exists, fall back on supervision time spans.

Return type `ndarray`

speakers_feature_mask(*min_speaker_dim=None*, *speaker_to_idx_map=None*, *use_alignment_if_exists=None*)

Return a matrix of per-speaker activity in a cut. The matrix shape is (num_speakers, num_frames), and its values are 0 for nonspeech **frames** and 1 for speech **frames** for each respective speaker.

This is somewhat inspired by the TS-VAD setup: https://arxiv.org/abs/2005.07272

Parameters

- **min_speaker_dim** (`Optional[int]`) – optional int, when specified it will enforce that the matrix shape is at least that value (useful for datasets like CHiME 6 where the number of speakers is always 4, but some cuts might have less speakers than that).
- **speaker_to_idx_map** (`Optional[Dict[str, int]]`) – optional dict mapping speaker names (strings) to their global indices (ints). Useful when you want to preserve the order of the speakers (e.g. speaker XYZ is always mapped to index 2)
- **use_alignment_if_exists** (`Optional[str]`) – optional str, key for alignment type to use for generating the mask. If not exists, fall back on supervision time spans.

Return type `ndarray`

split(*timestamp*)

Split a cut into two cuts at `timestamp`, which is measured from the start of the cut. For example, a [0s - 10s] cut split at 4s yields:

- left cut [0s - 4s]
- right cut [4s - 10s]

Return type `Tuple[Cut, Cut]`

supervisions_audio_mask(*use_alignment_if_exists=None*)
Return a 1D numpy array with value 1 for **samples** covered by at least one supervision, and 0 for **samples** not covered by any supervision.

Parameters **use_alignment_if_exists** (Optional[str]) – optional str, key for alignment type to use for generating the mask. If not exists, fall back on supervision time spans.

Return type ndarray

supervisions_feature_mask(*use_alignment_if_exists=None*)
Return a 1D numpy array with value 1 for **frames** covered by at least one supervision, and 0 for **frames** not covered by any supervision.

Parameters **use_alignment_if_exists** (Optional[str]) – optional str, key for alignment type to use for generating the mask. If not exists, fall back on supervision time spans.

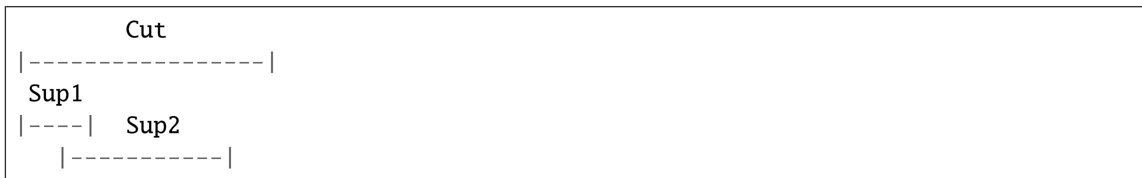
Return type ndarray

to_dict()

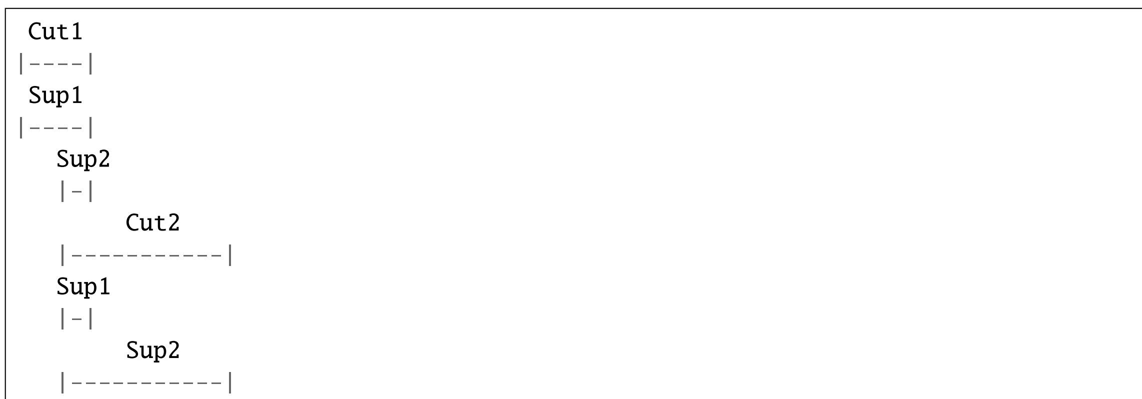
Return type dict

trim_to_supervisions(*keep_overlapping=True*, *min_duration=None*, *context_direction='center'*)
Splits the current *Cut* into as many cuts as there are supervisions (*SupervisionSegment*). These cuts have identical start times and durations as the supervisions. When there are overlapping supervisions, they can be kept or discarded via keep_overlapping flag.

For example, the following cut:



is transformed into two cuts:



Parameters

- **keep_overlapping** (bool) – when False, it will discard parts of other supervisions that overlap with the main supervision. In the illustration above, it would discard Sup2 in Cut1 and Sup1 in Cut2. In this mode, we guarantee that there will always be exactly one supervision per cut.

- **min_duration** (Optional[float]) – An optional duration in seconds; specifying this argument will extend the cuts that would have been shorter than min_duration with actual acoustic context in the recording/features. If there are supervisions present in the context, they are kept when keep_overlapping is true. If there is not enough context, the returned cut will be shorter than min_duration. If the supervision segment is longer than min_duration, the return cut will be longer.
- **context_direction** (Literal['center', 'left', 'right', 'random']) – Which direction should the cut be expanded towards to include context. The value of "center" implies equal expansion to left and right; random uniformly samples a value between "left" and "right".

Return type List[*Cut*]

Returns a list of cuts.

property trimmed_supervisions: List[*lhotse.supervision.SupervisionSegment*]

Return the supervisions in this Cut that have modified time boundaries so as not to exceed the Cut's start or end.

Note that when cut.supervisions is called, the supervisions may have negative start values that indicate the supervision actually begins before the cut, or end values that exceed the Cut's duration (it means the supervision continued in the original recording after the Cut's ending).

> **Caution:** For some tasks such as speech recognition (ASR), trimmed supervisions could result in corrupted training data. This is because a part of the transcript might actually reside outside of the cut.

Return type List[*SupervisionSegment*]

with_id(*id_*)

Return a copy of the Cut with a new ID.

Return type *Cut*

class lhotse.cut.**PaddingCut**(*id*, *duration*, *sampling_rate*, *feat_value*, *num_frames=None*, *num_features=None*, *frame_shift=None*, *num_samples=None*, *custom=None*)

PaddingCut is a dummy *Cut* that doesn't refer to actual recordings or features –it simply returns zero samples in the time domain and a specified features value in the feature domain. Its main role is to be appended to other cuts to make them evenly sized.

Please refer to the documentation of *Cut* to learn more about using cuts.

See also:

- *lhotse.cut.Cut*
- *lhotse.cut.MonoCut*
- *lhotse.cut.MixedCut*
- *lhotse.cut.CutSet*

id: str

duration: float

sampling_rate: int

feat_value: float

num_frames: Optional[int] = None

num_features: Optional[int] = None

frame_shift: Optional[float] = None

num_samples: Optional[int] = None

custom: Optional[dict] = None

property start: float

Return type float

property supervisions

property has_features: bool

Return type bool

property has_recording: bool

Return type bool

property recording_id: str

Return type str

load_features(**args*, ***kwargs*)

Return type Optional[ndarray]

load_audio(**args*, ***kwargs*)

Return type Optional[ndarray]

truncate(***, *offset=0.0*, *duration=None*, *keep_excessive_supervisions=True*, *preserve_id=False*, ***kwargs*)

Return type *PaddingCut*

extend_by(***, *duration*, *direction='both'*, *preserve_id=False*)

Return a new PaddingCut with region extended by the specified duration.

Parameters

- **duration** (float) – The duration by which to extend the cut.
- **direction** (str) – string, 'left', 'right' or 'both'. Determines whether the cut should be extended to the left, right or both sides. By default, the cut is extended by the specified duration on both sides.
- **preserve_id** (bool) – When True, preserves the cut ID from before padding. Otherwise, generates a new random ID (default).

Return type *PaddingCut*

Returns an extended PaddingCut.

pad(*duration=None*, *num_frames=None*, *num_samples=None*, *pad_feat_value=- 23.025850929940457*, *direction='right'*, *preserve_id=False*, *pad_value_dict=None*)

Return a new MixedCut, padded with zeros in the recording, and `pad_feat_value` in each feature bin.

The user can choose to pad either to a specific *duration*; a specific number of frames *max_frames*; or a specific number of samples *num_samples*. The three arguments are mutually exclusive.

Parameters

- **duration** (Optional[float]) – The cut's minimal duration after padding.
- **num_frames** (Optional[int]) – The cut's total number of frames after padding.
- **num_samples** (Optional[int]) – The cut's total number of samples after padding.
- **pad_feat_value** (float) – A float value that's used for padding the features. By default we assume a log-energy floor of approx. -23 (1e-10 after exp).
- **direction** (str) – string, 'left', 'right' or 'both'. Determines whether the padding is added before or after the cut.
- **preserve_id** (bool) – When True, preserves the cut ID from before padding. Otherwise, generates a new random ID (default).
- **pad_value_dict** (Optional[Dict[str, Union[int, float]]]) – Optional dict that specifies what value should be used for padding arrays in custom attributes.

Return type *Cut*

Returns a padded MixedCut if duration is greater than this cut's duration, otherwise self.

resample(*sampling_rate*, *affix_id=False*)

Return a new MonoCut that will lazily resample the audio while reading it. This operation will drop the feature manifest, if attached. It does not affect the supervision.

Parameters

- **sampling_rate** (int) – The new sampling rate.
- **affix_id** (bool) – Should we modify the ID (useful if both versions of the same cut are going to be present in a single manifest).

Return type *PaddingCut*

Returns a modified copy of the current MonoCut.

perturb_speed(*factor*, *affix_id=True*)

Return a new PaddingCut that will "mimic" the effect of speed perturbation on duration and num_samples.

Parameters

- **factor** (float) – The speed will be adjusted this many times (e.g. factor=1.1 means 1.1x faster).
- **affix_id** (bool) – When true, we will modify the PaddingCut.id field by affixing it with "_sp{factor}".

Return type *PaddingCut*

Returns a modified copy of the current PaddingCut.

perturb_tempo(*factor*, *affix_id=True*)

Return a new PaddingCut that will "mimic" the effect of tempo perturbation on duration and num_samples.

Compared to speed perturbation, tempo preserves pitch. :type factor: float :param factor: The tempo will be adjusted this many times (e.g. factor=1.1 means 1.1x faster). :type affix_id: bool :param affix_id: When true, we will modify the PaddingCut.id field

by affixing it with "_tp{factor}".

Return type *PaddingCut*

Returns a modified copy of the current PaddingCut.

perturb_volume(*factor*, *affix_id=True*)
Return a new PaddingCut that will "mimic" the effect of volume perturbation on amplitude of samples.

Parameters

- **factor** (float) – The volume will be adjusted this many times (e.g. factor=1.1 means 1.1x louder).
- **affix_id** (bool) – When true, we will modify the PaddingCut.id field by affixing it with "_vp{factor}".

Return type *PaddingCut*

Returns a modified copy of the current PaddingCut.

reverb_rir(*rir_recording*, *normalize_output=True*, *early_only=False*, *affix_id=True*, *rir_channels=[0]*)
Return a new PaddingCut that will "mimic" the effect of reverberation with impulse response on original samples.

Parameters

- **rir_recording** (*Recording*) – The impulse response to use for convolving.
- **normalize_output** (bool) – When true, output will be normalized to have energy as input.
- **early_only** (bool) – When true, only the early reflections (first 50 ms) will be used.
- **affix_id** (bool) – When true, we will modify the PaddingCut.id field by affixing it with "_rvb".
- **rir_channels** (List[int]) – The channels of the impulse response to use.

Return type *PaddingCut*

Returns a modified copy of the current PaddingCut.

drop_features()
Return a copy of the current *PaddingCut*, detached from features.

Return type *PaddingCut*

drop_recording()
Return a copy of the current *PaddingCut*, detached from recording.

Return type *PaddingCut*

drop_supervisions()
Return a copy of the current *PaddingCut*, detached from supervisions.

Return type *PaddingCut*

compute_and_store_features(*extractor*, **args*, ***kwargs*)
Returns a new PaddingCut with updates information about the feature dimension and number of feature frames, depending on the extractor properties.

Return type *Cut*

fill_supervision(**args*, ***kwargs*)
Just for consistency with :class`.MonoCut` and *MixedCut*.

Return type *PaddingCut*

map_supervisions(*transform_fn*)
Just for consistency with *MonoCut* and *MixedCut*.

Parameters **transform_fn** (Callable[[Any], Any]) – a dummy function that would be never called actually.

Return type *PaddingCut*

Returns the PaddingCut itself.

merge_supervisions(**args*, ***kwargs*)
Just for consistency with *MonoCut* and *MixedCut*.

Return type *PaddingCut*

Returns the PaddingCut itself.

filter_supervisions(*predicate*)
Just for consistency with *MonoCut* and *MixedCut*.

Parameters **predicate** (Callable[[*SupervisionSegment*], bool]) – A callable that accepts *SupervisionSegment* and returns bool

Return type *PaddingCut*

Returns a modified MonoCut

static from_dict(*data*)

Return type *PaddingCut*

with_features_path_prefix(*path*)

Return type *PaddingCut*

with_recording_path_prefix(*path*)

Return type *PaddingCut*

__init__(*id*, *duration*, *sampling_rate*, *feat_value*, *num_frames=None*, *num_features=None*, *frame_shift=None*, *num_samples=None*, *custom=None*)

append(*other*, *snr=None*, *preserve_id=None*)
Append the `other` Cut after the current Cut. Conceptually the same as `mix` but with an offset matching the current cuts length. Optionally scale down (positive SNR) or scale up (negative SNR) the `other` cut. Returns a MixedCut, which only keeps the information about the mix; actual mixing is performed during the call to `load_features`.

Parameters **preserve_id** (Optional[str]) – optional string ("left", "right"). When specified, append will preserve the cut ID of the left- or right-hand side argument. Otherwise, a new random ID is generated.

Return type *MixedCut*

compute_and_store_recording(*storage_path*, *augment_fn=None*)
Store this cut's waveform as audio recording to disk.

Parameters

- **storage_path** (Union[Path, str]) – The path to location where we will store the audio recordings.
- **augment_fn** (Optional[Callable[[ndarray, int], ndarray]]) – an optional callable used for audio augmentation. Be careful with the types of augmentations used: if they modify the start/end/duration times of the cut and its supervisions, you will end up with

incorrect supervision information when using this API. E.g. for speed perturbation, use `CutSet.perturb_speed()` instead.

Return type *MonoCut*

Returns a new MonoCut instance.

compute_features(*extractor*, *augment_fn=None*)
Compute the features from this cut. This cut has to be able to load audio.

Parameters

- **extractor** (*FeatureExtractor*) – a `FeatureExtractor` instance used to compute the features.
- **augment_fn** (`Optional[Callable[[ndarray, int], ndarray]]`) – optional `WavAugmenter` instance for audio augmentation.

Return type `ndarray`

Returns a numpy ndarray with the computed features.

cut_into_windows(*duration*, *hop=None*, *keep_excessive_supervisions=True*)
Return a list of shorter cuts, made by traversing this cut in windows of `duration` seconds by `hop` seconds.

The last window might have a shorter duration if there was not enough audio, so you might want to use either filter or pad the results.

Parameters

- **duration** (`float`) – Desired duration of the new cuts in seconds.
- **hop** (`Optional[float]`) – Shift between the windows in the new cuts in seconds.
- **keep_excessive_supervisions** (`bool`) – bool. When a cut is truncated in the middle of a supervision segment, should the supervision be kept.

Return type `List[`*Cut*`]`

Returns a list of cuts made from shorter duration windows.

property end: float

Return type `float`

index_supervisions(*index_mixed_tracks=False*, *keep_ids=None*)
Create a two-level index of supervision segments. It is a mapping from a Cut's ID to an interval tree that contains the supervisions of that Cut.

The interval tree can be efficiently queried for overlapping and/or enveloping segments. It helps speed up some operations on Cuts of very long recordings (1h+) that contain many supervisions.

Parameters

- **index_mixed_tracks** (`bool`) – Should the tracks of MixedCut's be indexed as additional, separate entries.
- **keep_ids** (`Optional[Set[str]]`) – If specified, we will only index the supervisions with the specified IDs.

Return type `Dict[str, IntervalTree]`

Returns a mapping from Cut ID to an interval tree of SupervisionSegments.

mix(*other*, *offset_other_by=0.0*, *allow_padding=False*, *snr=None*, *preserve_id=None*)
Refer to **:function:`~lhotse.cut.mix`** documentation.

Return type *MixedCut*

play_audio()
Display a Jupyter widget that allows to listen to the waveform. Works only in Jupyter notebook/lab or similar (e.g. Colab).

plot_alignment(*alignment_type='word'*)
Display the alignment on top of a spectrogram. Requires matplotlib to be installed.

plot_audio()
Display a plot of the waveform. Requires matplotlib to be installed.

plot_features()
Display the feature matrix as an image. Requires matplotlib to be installed.

save_audio(*storage_path*, *augment_fn=None*)
Store this cut's waveform as audio recording to disk.

Parameters

- **storage_path** (`Union[Path, str]`) – The path to location where we will store the audio recordings.
- **augment_fn** (`Optional[Callable[[ndarray, int], ndarray]]`) – an optional callable used for audio augmentation. Be careful with the types of augmentations used: if they modify the start/end/duration times of the cut and its supervisions, you will end up with incorrect supervision information when using this API. E.g. for speed perturbation, use `CutSet.perturb_speed()` instead.

Return type *MonoCut*

Returns a new MonoCut instance.

speakers_audio_mask(*min_speaker_dim=None*, *speaker_to_idx_map=None*, *use_alignment_if_exists=None*)
Return a matrix of per-speaker activity in a cut. The matrix shape is (num_speakers, num_samples), and its values are 0 for nonspeech **samples** and 1 for speech **samples** for each respective speaker.

This is somewhat inspired by the TS-VAD setup: https://arxiv.org/abs/2005.07272

Parameters

- **min_speaker_dim** (`Optional[int]`) – optional int, when specified it will enforce that the matrix shape is at least that value (useful for datasets like CHiME 6 where the number of speakers is always 4, but some cuts might have less speakers than that).
- **speaker_to_idx_map** (`Optional[Dict[str, int]]`) – optional dict mapping speaker names (strings) to their global indices (ints). Useful when you want to preserve the order of the speakers (e.g. speaker XYZ is always mapped to index 2)
- **use_alignment_if_exists** (`Optional[str]`) – optional str, key for alignment type to use for generating the mask. If not exists, fall back on supervision time spans.

Return type `ndarray`

speakers_feature_mask(*min_speaker_dim=None*, *speaker_to_idx_map=None*, *use_alignment_if_exists=None*)
Return a matrix of per-speaker activity in a cut. The matrix shape is (num_speakers, num_frames), and its values are 0 for nonspeech **frames** and 1 for speech **frames** for each respective speaker.

This is somewhat inspired by the TS-VAD setup: https://arxiv.org/abs/2005.07272

Parameters

- **min_speaker_dim** (Optional[int]) – optional int, when specified it will enforce that the matrix shape is at least that value (useful for datasets like CHiME 6 where the number of speakers is always 4, but some cuts might have less speakers than that).
- **speaker_to_idx_map** (Optional[Dict[str, int]]) – optional dict mapping speaker names (strings) to their global indices (ints). Useful when you want to preserve the order of the speakers (e.g. speaker XYZ is always mapped to index 2)
- **use_alignment_if_exists** (Optional[str]) – optional str, key for alignment type to use for generating the mask. If not exists, fall back on supervision time spans.

Return type ndarray

split(*timestamp*)

Split a cut into two cuts at timestamp, which is measured from the start of the cut. For example, a [0s - 10s] cut split at 4s yields:

- left cut [0s - 4s]
- right cut [4s - 10s]

Return type Tuple[Cut, Cut]

supervisions_audio_mask(*use_alignment_if_exists=None*)

Return a 1D numpy array with value 1 for **samples** covered by at least one supervision, and 0 for **samples** not covered by any supervision.

Parameters use_alignment_if_exists (Optional[str]) – optional str, key for alignment type to use for generating the mask. If not exists, fall back on supervision time spans.

Return type ndarray

supervisions_feature_mask(*use_alignment_if_exists=None*)

Return a 1D numpy array with value 1 for **frames** covered by at least one supervision, and 0 for **frames** not covered by any supervision.

Parameters use_alignment_if_exists (Optional[str]) – optional str, key for alignment type to use for generating the mask. If not exists, fall back on supervision time spans.

Return type ndarray

to_dict()

Return type dict

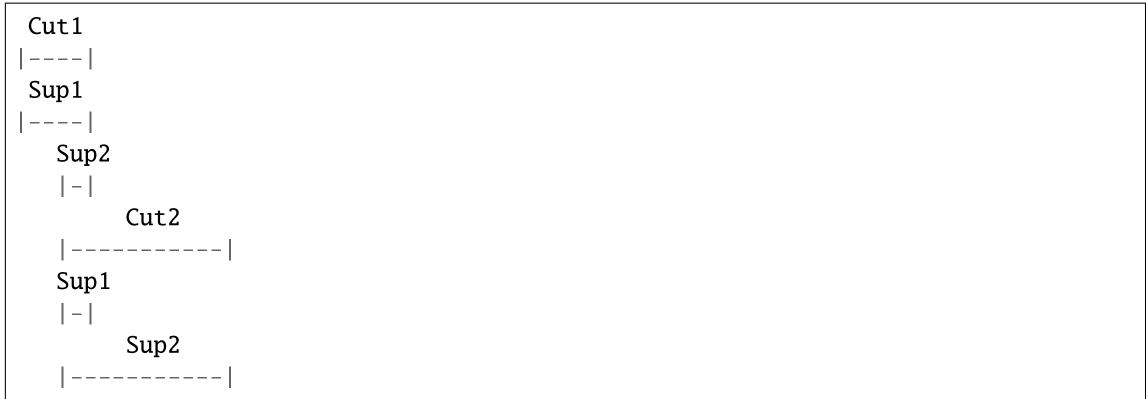
trim_to_supervisions(*keep_overlapping=True, min_duration=None, context_direction='center'*)

Splits the current Cut into as many cuts as there are supervisions (SupervisionSegment). These cuts have identical start times and durations as the supervisions. When there are overlapping supervisions, they can be kept or discarded via keep_overlapping flag.

For example, the following cut:

```
          Cut
|-----------------|
 Sup1
|----|  Sup2
   |-----------|
```

is transformed into two cuts:



Parameters

- **keep_overlapping** (`bool`) – when `False`, it will discard parts of other supervisions that overlap with the main supervision. In the illustration above, it would discard `Sup2` in `Cut1` and `Sup1` in `Cut2`. In this mode, we guarantee that there will always be exactly one supervision per cut.
- **min_duration** (`Optional[float]`) – An optional duration in seconds; specifying this argument will extend the cuts that would have been shorter than `min_duration` with actual acoustic context in the recording/features. If there are supervisions present in the context, they are kept when `keep_overlapping` is true. If there is not enough context, the returned cut will be shorter than `min_duration`. If the supervision segment is longer than `min_duration`, the return cut will be longer.
- **context_direction** (`Literal`['center', 'left', 'right', 'random']) – Which direction should the cut be expanded towards to include context. The value of "center" implies equal expansion to left and right; random uniformly samples a value between "left" and "right".

Return type `List[Cut]`

Returns a list of cuts.

property trimmed_supervisions: List[*lhotse.supervision.SupervisionSegment*]

Return the supervisions in this Cut that have modified time boundaries so as not to exceed the Cut's start or end.

Note that when `cut.supervisions` is called, the supervisions may have negative `start` values that indicate the supervision actually begins before the cut, or `end` values that exceed the Cut's duration (it means the supervision continued in the original recording after the Cut's ending).

> **Caution:** For some tasks such as speech recognition (ASR), trimmed supervisions could result in corrupted training data. This is because a part of the transcript might actually reside outside of the cut.

Return type `List[SupervisionSegment]`

with_id(*id_*)

Return a copy of the Cut with a new ID.

Return type *Cut*

features_type: Optional[str]

class lhotse.cut.**MixTrack**(*cut*, *offset=0.0*, *snr=None*)

Represents a single track in a mix of Cuts. Points to a specific MonoCut and holds information on how to mix it with other Cuts, relative to the first track in a mix.

cut: Union[*lhotse.cut.MonoCut*, *lhotse.cut.PaddingCut*]

offset: float = 0.0

snr: Optional[float] = None

static from_dict(*data*)

__init__(*cut*, *offset=0.0*, *snr=None*)

class lhotse.cut.**MixedCut**(*id*, *tracks*)

MixedCut is a *Cut* that actually consists of multiple other cuts. It can be interpreted as a multi-channel cut, but its primary purpose is to allow time-domain and feature-domain augmentation via mixing the training cuts with noise, music, and babble cuts. The actual mixing operations are performed on-the-fly.

Internally, *MixedCut* holds other cuts in multiple trakcs (*MixTrack*), each with its own offset and SNR that is relative to the first track.

Please refer to the documentation of *Cut* to learn more about using cuts.

In addition to methods available in *Cut*, *MixedCut* provides the methods to read all of its tracks audio and features as separate channels:

```
>>> cut = MixedCut(...)
>>> mono_features = cut.load_features()
>>> assert len(mono_features.shape) == 2
>>> multi_features = cut.load_features(mixed=False)
>>> # Now, the first dimension is the channel.
>>> assert len(multi_features.shape) == 3
```

See also:

- *lhotse.cut.Cut*
- *lhotse.cut.MonoCut*
- *lhotse.cut.CutSet*

id: str

tracks: List[*lhotse.cut.MixTrack*]

property supervisions: List[*lhotse.supervision.SupervisionSegment*]

Lists the supervisions of the underlying source cuts. Each segment start time will be adjusted by the track offset.

Return type List[*SupervisionSegment*]

property start: float

Return type float

property duration: float

Return type float

property has_features: bool

Return type bool

property has_recording: bool

Return type bool

property num_frames: Optional[int]

Return type Optional[int]

property frame_shift: Optional[float]

Return type Optional[float]

property sampling_rate: Optional[int]

Return type Optional[int]

property num_samples: Optional[int]

Return type Optional[int]

property num_features: Optional[int]

Return type Optional[int]

property features_type: Optional[str]

Return type Optional[str]

load_custom(*name*)

Load custom data as numpy array. The custom data is expected to have been stored in cuts `custom` field as an `Array` or `TemporalArray` manifest.

Note: It works with Array manifests stored via attribute assignments, e.g.: `cut.my_custom_data = Array(...)`.

Warning: For *MixedCut*, this will only work if the mixed cut consists of a single *MonoCut* and an arbitrary number of `PaddingCuts`. This is because it is generally undefined how to mix arbitrary arrays.

Parameters **name** (`str`) – name of the custom attribute.

Return type ndarray

Returns a numpy array with the data (after padding).

truncate(*, *offset=0.0*, *duration=None*, *keep_excessive_supervisions=True*, *preserve_id=False*, *_supervisions_index=None*)

Returns a new MixedCut that is a sub-region of the current MixedCut. This method truncates the underlying Cuts and modifies their offsets in the mix, as needed. Tracks that do not fit in the truncated cut are removed.

Note that no operation is done on the actual features - it's only during the call to load_features() when the actual changes happen (a subset of features is loaded).

Parameters

- **offset** (`float`) – float (seconds), controls the start of the new cut relative to the current MixedCut's start.
- **duration** (`Optional[float]`) – optional float (seconds), controls the duration of the resulting MixedCut. By default, the duration is (end of the cut before truncation) - (offset).
- **keep_excessive_supervisions** (`bool`) – bool. Since trimming may happen inside a SupervisionSegment, the caller has an option to either keep or discard such supervisions.

- **preserve_id** (bool) – bool. Should the truncated cut keep the same ID or get a new, random one.

Return type *Cut*

Returns a new MixedCut instance.

extend_by(*, *duration*, *direction='both'*, *preserve_id=False*)
This raises a ValueError since extending a MixedCut is not defined.

Parameters

- **duration** (float) – float (seconds), duration (in seconds) to extend the MixedCut.
- **direction** (str) – string, 'left', 'right' or 'both'. Determines whether to extend on the left, right, or both sides. If 'both', extend on both sides by the duration specified in *duration*.
- **preserve_id** (bool) – bool. Should the extended cut keep the same ID or get a new, random one.

Return type *MixedCut*

Returns a new MixedCut instance.

pad(*duration=None*, *num_frames=None*, *num_samples=None*, *pad_feat_value=- 23.025850929940457*, *direction='right'*, *preserve_id=False*, *pad_value_dict=None*)
Return a new MixedCut, padded with zeros in the recording, and `pad_feat_value` in each feature bin.

The user can choose to pad either to a specific *duration*; a specific number of frames *max_frames*; or a specific number of samples *num_samples*. The three arguments are mutually exclusive.

Parameters

- **duration** (Optional[float]) – The cut's minimal duration after padding.
- **num_frames** (Optional[int]) – The cut's total number of frames after padding.
- **num_samples** (Optional[int]) – The cut's total number of samples after padding.
- **pad_feat_value** (float) – A float value that's used for padding the features. By default we assume a log-energy floor of approx. -23 (1e-10 after exp).
- **direction** (str) – string, 'left', 'right' or 'both'. Determines whether the padding is added before or after the cut.
- **preserve_id** (bool) – When `True`, preserves the cut ID from before padding. Otherwise, generates a new random ID (default).
- **pad_value_dict** (Optional[Dict[str, Union[int, float]]]) – Optional dict that specifies what value should be used for padding arrays in custom attributes.

Return type *Cut*

Returns a padded MixedCut if duration is greater than this cut's duration, otherwise `self`.

resample(*sampling_rate*, *affix_id=False*)
Return a new `MixedCut` that will lazily resample the audio while reading it. This operation will drop the feature manifest, if attached. It does not affect the supervision.

Parameters

- **sampling_rate** (int) – The new sampling rate.
- **affix_id** (bool) – Should we modify the ID (useful if both versions of the same cut are going to be present in a single manifest).

Return type *MixedCut*

Returns a modified copy of the current `MixedCut`.

`perturb_speed`(*factor*, *affix_id=True*)
Return a new `MixedCut` that will lazily perturb the speed while loading audio. The `num_samples`, `start` and `duration` fields of the underlying Cuts (and their Recordings and SupervisionSegments) are updated to reflect the shrinking/extending effect of speed. We are also updating the offsets of all underlying tracks.

Parameters

- **factor** (`float`) – The speed will be adjusted this many times (e.g. factor=1.1 means 1.1x faster).
- **affix_id** (`bool`) – When true, we will modify the `MixedCut.id` field by affixing it with "_sp{factor}".

Return type *MixedCut*

Returns a modified copy of the current `MixedCut`.

`perturb_tempo`(*factor*, *affix_id=True*)
Return a new `MixedCut` that will lazily perturb the tempo while loading audio.

Compared to speed perturbation, tempo preserves pitch. The `num_samples`, `start` and `duration` fields of the underlying Cuts (and their Recordings and SupervisionSegments) are updated to reflect the shrinking/extending effect of tempo. We are also updating the offsets of all underlying tracks.

Parameters

- **factor** (`float`) – The tempo will be adjusted this many times (e.g. factor=1.1 means 1.1x faster).
- **affix_id** (`bool`) – When true, we will modify the `MixedCut.id` field by affixing it with "_tp{factor}".

Return type *MixedCut*

Returns a modified copy of the current `MixedCut`.

`perturb_volume`(*factor*, *affix_id=True*)
Return a new `MixedCut` that will lazily perturb the volume while loading audio. Recordings of the underlying Cuts are updated to reflect volume change.

Parameters

- **factor** (`float`) – The volume will be adjusted this many times (e.g. factor=1.1 means 1.1x louder).
- **affix_id** (`bool`) – When true, we will modify the `MixedCut.id` field by affixing it with "_vp{factor}".

Return type *MixedCut*

Returns a modified copy of the current `MixedCut`.

`reverb_rir`(*rir_recording*, *normalize_output=True*, *early_only=False*, *affix_id=True*, *rir_channels=[0]*)
Return a new `MixedCut` that will convolve the audio with the provided impulse response.

Parameters

- **rir_recording** (*Recording*) – The impulse response to use for convolving.
- **normalize_output** (`bool`) – When true, output will be normalized to have energy as input.
- **early_only** (`bool`) – When true, only the early reflections (first 50 ms) will be used.

- **affix_id** (bool) – When true, we will modify the MixedCut.id field by affixing it with "_rvb".
- **rir_channels** (List[int]) – The channels of the impulse response to use. By default, first channel is used. If only one channel is specified, all tracks will be convolved with this channel. If a list is provided, it must contain as many channels as there are tracks such that each track will be convolved with one of the specified channels.

Return type *MixedCut*

Returns a modified copy of the current MixedCut.

load_features(*mixed=True*)

Loads the features of the source cuts and mixes them on-the-fly.

Parameters **mixed** (bool) – when True (default), returns a 2D array of features mixed in the feature domain. Otherwise returns a 3D array with the first dimension equal to the number of tracks.

Return type Optional[ndarray]

Returns A numpy ndarray with features and with shape (num_frames, num_features), or (num_tracks, num_frames, num_features)

load_audio(*mixed=True*)

Loads the audios of the source cuts and mix them on-the-fly.

Parameters **mixed** (bool) – When True (default), returns a mono mix of the underlying tracks. Otherwise returns a numpy array with the number of channels equal to the number of tracks.

Return type Optional[ndarray]

Returns A numpy ndarray with audio samples and with shape (num_channels, num_samples)

plot_tracks_features()

Display the feature matrix as an image. Requires matplotlib to be installed.

plot_tracks_audio()

Display plots of the individual tracks' waveforms. Requires matplotlib to be installed.

drop_features()

Return a copy of the current *MixedCut*, detached from features.

Return type *MixedCut*

drop_recording()

Return a copy of the current *MixedCut*, detached from recording.

Return type *MixedCut*

drop_supervisions()

Return a copy of the current *MixedCut*, detached from supervisions.

Return type *MixedCut*

compute_and_store_features(*extractor, storage, augment_fn=None, mix_eagerly=True*)

Compute the features from this cut, store them on disk, and create a new *MonoCut* object with the feature manifest attached. This cut has to be able to load audio.

Parameters

- **extractor** (*FeatureExtractor*) – a FeatureExtractor instance used to compute the features.

- **storage** (*FeaturesWriter*) – a FeaturesWriter instance used to store the features.
- **augment_fn** (Optional[Callable[[ndarray, int], ndarray]]) – an optional callable used for audio augmentation.
- **mix_eagerly** (bool) – when False, extract and store the features for each track separately, and mix them dynamically when loading the features. When True, mix the audio first and store the mixed features, returning a new MonoCut instance with the same ID. The returned MonoCut will not have a Recording attached.

Return type *Cut*

Returns a new MonoCut instance if mix_eagerly is True, or returns self with each of the tracks containing the Features manifests.

fill_supervision(*add_empty=True*, *shrink_ok=False*)

Fills the whole duration of a cut with a supervision segment.

If the cut has one supervision, its start is set to 0 and duration is set to cut.duration. Note: this may either expand a supervision that was shorter than a cut, or shrink a supervision that exceeds the cut.

If there are no supervisions, we will add an empty one when add_empty==True, otherwise we won't change anything.

If there are two or more supervisions, we will raise an exception.

Note: For *MixedCut*, we expect that only one track contains a supervision. That supervision will be expanded to cover the full MixedCut's duration.

Parameters

- **add_empty** (bool) – should we add an empty supervision with identical time bounds as the cut.
- **shrink_ok** (bool) – should we raise an error if a supervision would be shrank as a result of calling this method.

Return type *MixedCut*

map_supervisions(*transform_fn*)

Modify the SupervisionSegments by *transform_fn* of this MixedCut.

Parameters **transform_fn** (Callable[[*SupervisionSegment*], *SupervisionSegment*]) – a function that modifies a supervision as an argument.

Return type *Cut*

Returns a modified MixedCut.

merge_supervisions(*custom_merge_fn=None*)

Return a copy of the cut that has all of its supervisions merged into a single segment.

The new start is the start of the earliest superivion, and the new duration is a minimum spanning duration for all the supervisions.

The text fields are concatenated with a whitespace, and all other string fields (including IDs) are prefixed with "cat#" and concatenated with a hash symbol "#". This is also applied to custom fields. Fields with a None value are omitted.

Note: If you're using individual tracks of a mixed cut, note that this transform drops all the supervisions in individual tracks and assigns the merged supervision in the first *MonoCut* found in `self.tracks`.

Parameters **custom_merge_fn** (`Optional[Callable[[str, Iterable[Any]], Any]]`) – a function that will be called to merge custom fields values. We expect `custom_merge_fn` to handle all possible custom keys. When not provided, we will treat all custom values as strings. It will be called roughly like: `custom_merge_fn(custom_key, [s.custom[custom_key] for s in sups])`

Return type *MixedCut*

filter_supervisions(*predicate*)

Modify cut to store only supervisions accepted by *predicate*

Example:

```
>>> cut = cut.filter_supervisions(lambda s: s.id in supervision_ids)
>>> cut = cut.filter_supervisions(lambda s: s.duration < 5.0)
>>> cut = cut.filter_supervisions(lambda s: s.text is not None)
```

Parameters **predicate** (`Callable[[SupervisionSegment], bool]`) – A callable that accepts *SupervisionSegment* and returns bool

Return type *Cut*

Returns a modified MonoCut

static **from_dict**(*data*)

Return type *MixedCut*

with_features_path_prefix(*path*)

Return type *MixedCut*

with_recording_path_prefix(*path*)

Return type *MixedCut*

__init__(*id*, *tracks*)

append(*other*, *snr=None*, *preserve_id=None*)

Append the `other` Cut after the current Cut. Conceptually the same as `mix` but with an offset matching the current cuts length. Optionally scale down (positive SNR) or scale up (negative SNR) the `other` cut. Returns a MixedCut, which only keeps the information about the mix; actual mixing is performed during the call to `load_features`.

Parameters **preserve_id** (`Optional[str]`) – optional string ("left", "right"). When specified, append will preserve the cut ID of the left- or right-hand side argument. Otherwise, a new random ID is generated.

Return type *MixedCut*

compute_and_store_recording(*storage_path*, *augment_fn=None*)

Store this cut's waveform as audio recording to disk.

Parameters

- **storage_path** (Union[Path, str]) – The path to location where we will store the audio recordings.
- **augment_fn** (Optional[Callable[[ndarray, int], ndarray]]) – an optional callable used for audio augmentation. Be careful with the types of augmentations used: if they modify the start/end/duration times of the cut and its supervisions, you will end up with incorrect supervision information when using this API. E.g. for speed perturbation, use `CutSet.perturb_speed()` instead.

Return type *MonoCut*

Returns a new MonoCut instance.

compute_features(*extractor*, *augment_fn=None*)
Compute the features from this cut. This cut has to be able to load audio.

Parameters

- **extractor** (*FeatureExtractor*) – a `FeatureExtractor` instance used to compute the features.
- **augment_fn** (Optional[Callable[[ndarray, int], ndarray]]) – optional `WavAugmenter` instance for audio augmentation.

Return type ndarray

Returns a numpy ndarray with the computed features.

cut_into_windows(*duration*, *hop=None*, *keep_excessive_supervisions=True*)
Return a list of shorter cuts, made by traversing this cut in windows of `duration` seconds by `hop` seconds.

The last window might have a shorter duration if there was not enough audio, so you might want to use either filter or pad the results.

Parameters

- **duration** (float) – Desired duration of the new cuts in seconds.
- **hop** (Optional[float]) – Shift between the windows in the new cuts in seconds.
- **keep_excessive_supervisions** (bool) – bool. When a cut is truncated in the middle of a supervision segment, should the supervision be kept.

Return type List[*Cut*]

Returns a list of cuts made from shorter duration windows.

property end: float

Return type float

index_supervisions(*index_mixed_tracks=False*, *keep_ids=None*)
Create a two-level index of supervision segments. It is a mapping from a Cut's ID to an interval tree that contains the supervisions of that Cut.

The interval tree can be efficiently queried for overlapping and/or enveloping segments. It helps speed up some operations on Cuts of very long recordings (1h+) that contain many supervisions.

Parameters

- **index_mixed_tracks** (bool) – Should the tracks of MixedCut's be indexed as additional, separate entries.

- **keep_ids** (Optional[Set[str]]) – If specified, we will only index the supervisions with the specified IDs.

Return type Dict[str, IntervalTree]

Returns a mapping from Cut ID to an interval tree of SupervisionSegments.

mix(*other*, *offset_other_by=0.0*, *allow_padding=False*, *snr=None*, *preserve_id=None*)
Refer to **:function:`~lhotse.cut.mix`** documentation.

Return type *MixedCut*

play_audio()
Display a Jupyter widget that allows to listen to the waveform. Works only in Jupyter notebook/lab or similar (e.g. Colab).

plot_alignment(*alignment_type='word'*)
Display the alignment on top of a spectrogram. Requires matplotlib to be installed.

plot_audio()
Display a plot of the waveform. Requires matplotlib to be installed.

plot_features()
Display the feature matrix as an image. Requires matplotlib to be installed.

save_audio(*storage_path*, *augment_fn=None*)
Store this cut's waveform as audio recording to disk.

Parameters

- **storage_path** (Union[Path, str]) – The path to location where we will store the audio recordings.
- **augment_fn** (Optional[Callable[[ndarray, int], ndarray]]) – an optional callable used for audio augmentation. Be careful with the types of augmentations used: if they modify the start/end/duration times of the cut and its supervisions, you will end up with incorrect supervision information when using this API. E.g. for speed perturbation, use CutSet.perturb_speed() instead.

Return type *MonoCut*

Returns a new MonoCut instance.

speakers_audio_mask(*min_speaker_dim=None*, *speaker_to_idx_map=None*, *use_alignment_if_exists=None*)
Return a matrix of per-speaker activity in a cut. The matrix shape is (num_speakers, num_samples), and its values are 0 for nonspeech **samples** and 1 for speech **samples** for each respective speaker.

This is somewhat inspired by the TS-VAD setup: https://arxiv.org/abs/2005.07272

Parameters

- **min_speaker_dim** (Optional[int]) – optional int, when specified it will enforce that the matrix shape is at least that value (useful for datasets like CHiME 6 where the number of speakers is always 4, but some cuts might have less speakers than that).
- **speaker_to_idx_map** (Optional[Dict[str, int]]) – optional dict mapping speaker names (strings) to their global indices (ints). Useful when you want to preserve the order of the speakers (e.g. speaker XYZ is always mapped to index 2)
- **use_alignment_if_exists** (Optional[str]) – optional str, key for alignment type to use for generating the mask. If not exists, fall back on supervision time spans.

Return type ndarray

speakers_feature_mask(*min_speaker_dim=None*, *speaker_to_idx_map=None*, *use_alignment_if_exists=None*)

Return a matrix of per-speaker activity in a cut. The matrix shape is (num_speakers, num_frames), and its values are 0 for nonspeech **frames** and 1 for speech **frames** for each respective speaker.

This is somewhat inspired by the TS-VAD setup: https://arxiv.org/abs/2005.07272

Parameters

- **min_speaker_dim** (Optional[int]) – optional int, when specified it will enforce that the matrix shape is at least that value (useful for datasets like CHiME 6 where the number of speakers is always 4, but some cuts might have less speakers than that).
- **speaker_to_idx_map** (Optional[Dict[str, int]]) – optional dict mapping speaker names (strings) to their global indices (ints). Useful when you want to preserve the order of the speakers (e.g. speaker XYZ is always mapped to index 2)
- **use_alignment_if_exists** (Optional[str]) – optional str, key for alignment type to use for generating the mask. If not exists, fall back on supervision time spans.

Return type ndarray

split(*timestamp*)

Split a cut into two cuts at `timestamp`, which is measured from the start of the cut. For example, a [0s - 10s] cut split at 4s yields:

- left cut [0s - 4s]
- right cut [4s - 10s]

Return type Tuple[*Cut*, *Cut*]

supervisions_audio_mask(*use_alignment_if_exists=None*)

Return a 1D numpy array with value 1 for **samples** covered by at least one supervision, and 0 for **samples** not covered by any supervision.

Parameters **use_alignment_if_exists** (Optional[str]) – optional str, key for alignment type to use for generating the mask. If not exists, fall back on supervision time spans.

Return type ndarray

supervisions_feature_mask(*use_alignment_if_exists=None*)

Return a 1D numpy array with value 1 for **frames** covered by at least one supervision, and 0 for **frames** not covered by any supervision.

Parameters **use_alignment_if_exists** (Optional[str]) – optional str, key for alignment type to use for generating the mask. If not exists, fall back on supervision time spans.

Return type ndarray

to_dict()

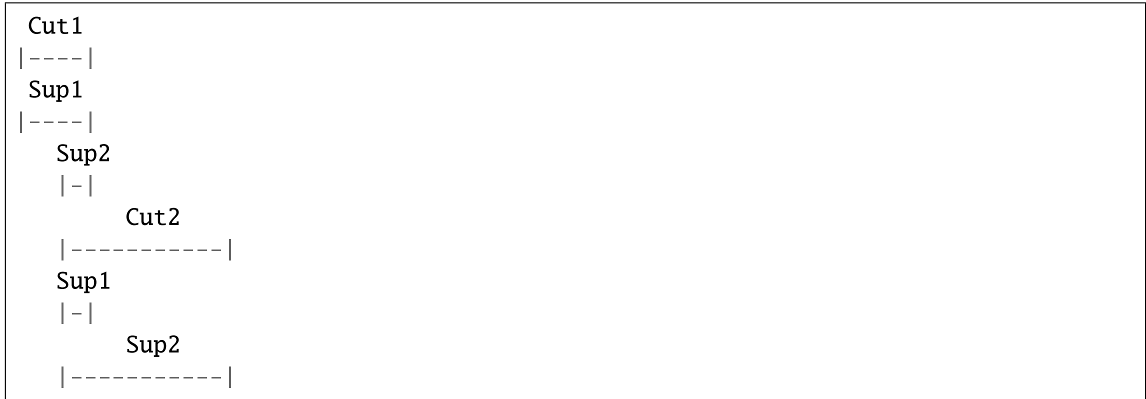
Return type dict

trim_to_supervisions(*keep_overlapping=True*, *min_duration=None*, *context_direction='center'*)

Splits the current *Cut* into as many cuts as there are supervisions (*SupervisionSegment*). These cuts have identical start times and durations as the supervisions. When there are overlapping supervisions, they can be kept or discarded via `keep_overlapping` flag.

For example, the following cut:

is transformed into two cuts:



Parameters

- **keep_overlapping** (bool) – when False, it will discard parts of other supervisions that overlap with the main supervision. In the illustration above, it would discard Sup2 in Cut1 and Sup1 in Cut2. In this mode, we guarantee that there will always be exactly one supervision per cut.
- **min_duration** (Optional[float]) – An optional duration in seconds; specifying this argument will extend the cuts that would have been shorter than min_duration with actual acoustic context in the recording/features. If there are supervisions present in the context, they are kept when keep_overlapping is true. If there is not enough context, the returned cut will be shorter than min_duration. If the supervision segment is longer than min_duration, the return cut will be longer.
- **context_direction** (Literal['center', 'left', 'right', 'random']) – Which direction should the cut be expanded towards to include context. The value of "center" implies equal expansion to left and right; random uniformly samples a value between "left" and "right".

Return type List[*Cut*]

Returns a list of cuts.

property trimmed_supervisions: List[*lhotse.supervision.SupervisionSegment*]

Return the supervisions in this Cut that have modified time boundaries so as not to exceed the Cut's start or end.

Note that when cut.supervisions is called, the supervisions may have negative start values that indicate the supervision actually begins before the cut, or end values that exceed the Cut's duration (it means the supervision continued in the original recording after the Cut's ending).

Caution: For some tasks such as speech recognition (ASR), trimmed supervisions could result in corrupted training data. This is because a part of the transcript might actually reside outside of the cut.

Return type List[*SupervisionSegment*]

with_id(*id_*)
Return a copy of the Cut with a new ID.

Return type `Cut`

class `lhotse.cut.`**CutSet**(*cuts=None*)
`CutSet` represents a collection of cuts, indexed by cut IDs. CutSet ties together all types of data – audio, features and supervisions, and is suitable to represent training/dev/test sets.

Note: `CutSet` is the basic building block of PyTorch-style Datasets for speech/audio processing tasks.

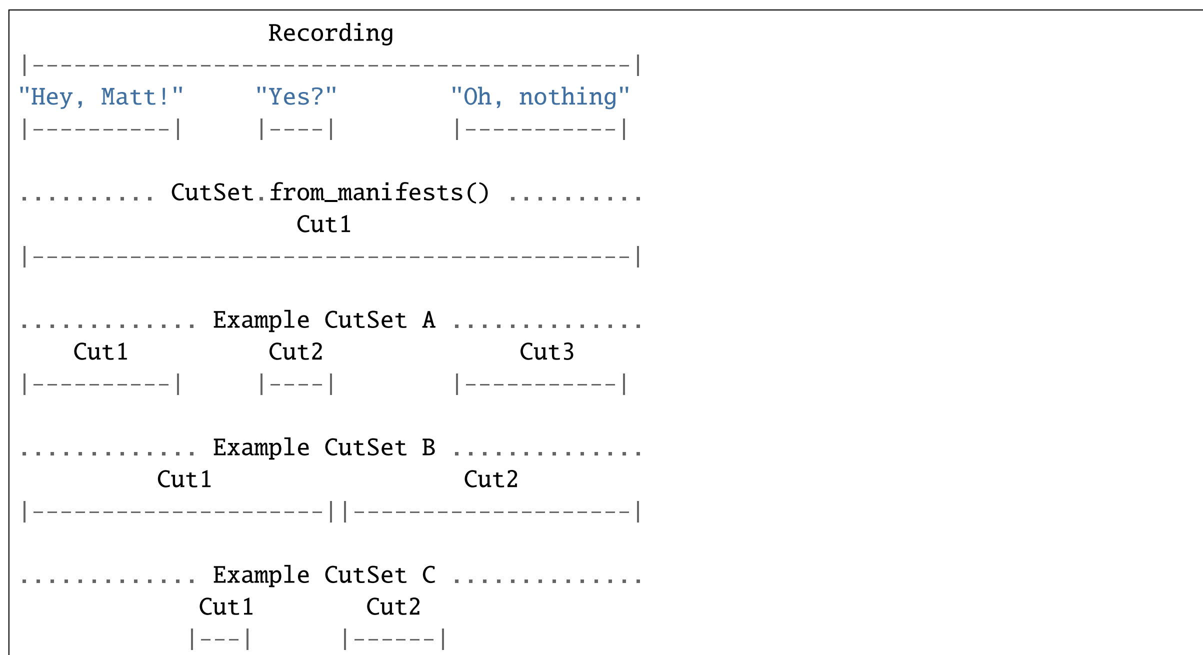
When coming from Kaldi, there is really no good equivalent – the closest concept may be Kaldi's "egs" for training neural networks, which are chunks of feature matrices and corresponding alignments used respectively as inputs and supervisions. `CutSet` is different because it provides you with all kinds of metadata, and you can select just the interesting bits to feed them to your models.

`CutSet` can be created from any combination of `RecordingSet`, `SupervisionSet`, and `FeatureSet` with `lhotse.cut.CutSet.from_manifests()`:

```
>>> from lhotse import CutSet
>>> cuts = CutSet.from_manifests(recordings=my_recording_set)
>>> cuts2 = CutSet.from_manifests(features=my_feature_set)
>>> cuts3 = CutSet.from_manifests(
...     recordings=my_recording_set,
...     features=my_feature_set,
...     supervisions=my_supervision_set,
... )
```

When creating a `CutSet` with `CutSet.from_manifests()`, the resulting cuts will have the same duration as the input recordings or features. For long recordings, it is not viable for training. We provide several methods to transform the cuts into shorter ones.

Consider the following scenario:



```
                  Recording
|-------------------------------------------|
"Hey, Matt!"     "Yes?"        "Oh, nothing"
|----------|      |----|         |-----------|

.......... CutSet.from_manifests() ..........
                    Cut1
|-------------------------------------------|

............. Example CutSet A ..............
    Cut1          Cut2              Cut3
|----------|      |----|         |-----------|

............. Example CutSet B ..............
          Cut1                  Cut2
|---------------------||--------------------|

............. Example CutSet C ..............
             Cut1        Cut2
            |---|       |------|
```

The CutSet's A, B and C can be created like:

```
>>> cuts_A = cuts.trim_to_supervisions()
>>> cuts_B = cuts.cut_into_windows(duration=5.0)
>>> cuts_C = cuts.trim_to_unsupervised_segments()
```

Note: Some operations support parallel execution via an optional `num_jobs` parameter. By default, all processing is single-threaded.

Caution: Operations on cut sets are not mutating – they return modified copies of *CutSet* objects, leaving the original object unmodified (and all of its cuts are also unmodified).

CutSet can be stored and read from JSON, JSONL, etc. and supports optional gzip compression:

```
>>> cuts.to_file('cuts.jsonl.gz')
>>> cuts4 = CutSet.from_file('cuts.jsonl.gz')
```

It behaves similarly to a `dict`:

```
>>> 'rec1-1-0' in cuts
True
>>> cut = cuts['rec1-1-0']
>>> for cut in cuts:
>>>    pass
>>> len(cuts)
127
```

CutSet has some convenience properties and methods to gather information about the dataset:

```
>>> ids = list(cuts.ids)
>>> speaker_id_set = cuts.speakers
>>> # The following prints a message:
>>> cuts.describe()
Cuts count: 547
Total duration (hours): 326.4
Speech duration (hours): 79.6 (24.4%)
***
Duration statistics (seconds):
mean    2148.0
std      870.9
min      477.0
25%     1523.0
50%     2157.0
75%     2423.0
max     5415.0
dtype: float64
```

Manipulation examples:

```
>>> longer_than_5s = cuts.filter(lambda c: c.duration > 5)
>>> first_100 = cuts.subset(first=100)
```

(continues on next page)

(continued from previous page)

```
>>> split_into_4 = cuts.split(num_splits=4)
>>> shuffled = cuts.shuffle()
>>> random_sample = cuts.sample(n_cuts=10)
>>> new_ids = cuts.modify_ids(lambda c: c.id + '-newid')
```

These operations can be composed to implement more complex operations, e.g. bucketing by duration:

```
>>> buckets = cuts.sort_by_duration().split(num_splits=30)
```

Cuts in a `CutSet` can be detached from parts of their metadata:

```
>>> cuts_no_feat = cuts.drop_features()
>>> cuts_no_rec = cuts.drop_recordings()
>>> cuts_no_sup = cuts.drop_supervisions()
```

Sometimes specific sorting patterns are useful when a small CutSet represents a mini-batch:

```
>>> cuts = cuts.sort_by_duration(ascending=False)
>>> cuts = cuts.sort_like(other_cuts)
```

`CutSet` offers some batch processing operations:

```
>>> cuts = cuts.pad(num_frames=300)  # or duration=30.0
>>> cuts = cuts.truncate(max_duration=30.0, offset_type='start')  # truncate from␣
↪start to 30.0s
>>> cuts = cuts.mix(other_cuts, snr=[10, 30], mix_prob=0.5)
```

`CutSet` supports lazy data augmentation/transformation methods which require adjusting some information in the manifest (e.g., `num_samples` or `duration`). Note that in the following examples, the audio is untouched – the operations are stored in the manifest, and executed upon reading the audio:

```
>>> cuts_sp = cuts.perturb_speed(factor=1.1)
>>> cuts_vp = cuts.perturb_volume(factor=2.)
>>> cuts_24k = cuts.resample(24000)
>>> cuts_rvb = cuts.reverb_rir(rir_recordings)
```

Caution: If the `CutSet` contained `Features` manifests, they will be detached after performing audio augmentations such as `CutSet.perturb_speed()`, `CutSet.resample()`, `CutSet.perturb_volume()`, or `CutSet.reverb_rir()`.

`CutSet` offers parallel feature extraction capabilities (see *meth*:.CutSet.compute_and_store_features: for details), and can be used to estimate global mean and variance:

```
>>> from lhotse import Fbank
>>> cuts = CutSet()
>>> cuts = cuts.compute_and_store_features(
...     extractor=Fbank(),
...     storage_path='/data/feats',
...     num_jobs=4
... )
>>> mvn_stats = cuts.compute_global_feature_stats('/data/features/mvn_stats.pkl',␣
↪max_cuts=10000)
```

See also:

- *Cut*

__init__(*cuts=None*)

property data: Union[Dict[str, *lhotse.cut.Cut*], Iterable[*lhotse.cut.Cut*]]
Alias property for `self.cuts`

Return type Union[Dict[str, *Cut*], Iterable[*Cut*]]

property mixed_cuts: Dict[str, *lhotse.cut.MixedCut*]

Return type Dict[str, *MixedCut*]

property simple_cuts: Dict[str, *lhotse.cut.MonoCut*]

Return type Dict[str, *MonoCut*]

property ids: Iterable[str]

Return type Iterable[str]

property speakers: FrozenSet[str]

Return type FrozenSet[str]

static from_cuts(*cuts*)

Return type *CutSet*

static from_items(*cuts*)
Function to be implemented by every sub-class of this mixin. It's expected to create a sub-class instance out of an iterable of items that are held by the sub-class (e.g., `CutSet.from_items(iterable_of_cuts)`).

Return type *CutSet*

static from_manifests(*recordings=None*, *supervisions=None*, *features=None*, *output_path=None*, *random_ids=False*, *lazy=False*)
Create a CutSet from any combination of supervision, feature and recording manifests. At least one of `recordings` or `features` is required.

The created cuts will be of type *MonoCut*, even when the recordings have multiple channels. The *MonoCut* boundaries correspond to those found in the `features`, when available, otherwise to those found in the `recordings`.

When `supervisions` are provided, we'll be searching them for matching recording IDs and attaching to created cuts, assuming they are fully within the cut's time span.

Parameters

- **recordings** (Optional[*RecordingSet*]) – an optional *RecordingSet* manifest.
- **supervisions** (Optional[*SupervisionSet*]) – an optional *SupervisionSet* manifest.
- **features** (Optional[*FeatureSet*]) – an optional *FeatureSet* manifest.
- **output_path** (Union[Path, str, None]) – an optional path where the *CutSet* is stored.
- **random_ids** (bool) – boolean, should the cut IDs be randomized. By default, use the recording ID with a loop index and a channel idx, i.e. "{recording_id}-{idx}-{channel}")
- **lazy** (bool) – boolean, when `True`, output_path must be provided

Return type *CutSet*

Returns a new *CutSet* instance.

static from_dicts(*data*)

Return type *CutSet*

static from_webdataset(*path*, ***wds_kwargs*)

Provides the ability to read Lhotse objects from a WebDataset tarball (or a collection of them, i.e., shards) sequentially, without reading the full contents into memory. It also supports passing a list of paths, or WebDataset-style pipes.

CutSets stored in this format are potentially much faster to read from due to sequential I/O (we observed speedups of 50-100x vs random-read mechanisms).

Since this mode does not support random access reads, some methods of CutSet might not work properly (e.g. `len()`).

The behaviour of the underlying `WebDataset` instance can be customized by providing its kwargs directly to the constructor of this class. For details, see `lhotse.dataset.webdataset.mini_webdataset()` documentation.

Examples

Read manifests and data from a single tarball:

```
>>> cuts = CutSet.from_webdataset("data/cuts-train.tar")
```

Read manifests and data from a multiple tarball shards:

```
>>> cuts = CutSet.from_webdataset("data/shard-{000000..004126}.tar")
>>> # alternatively
>>> cuts = CutSet.from_webdataset(["data/shard-000000.tar", "data/shard-000001.
↪tar", ...])
```

Read manifests and data from shards in cloud storage (here AWS S3 via AWS CLI):

```
>>> cuts = CutSet.from_webdataset("pipe:aws s3 cp data/shard-{000000..004126}.
↪tar -")
```

Read manifests and data from shards which are split between PyTorch DistributeDataParallel nodes and dataloading workers, with shard-level shuffling enabled:

```
>>> cuts = CutSet.from_webdataset(
...     "data/shard-{000000..004126}.tar",
...     split_by_worker=True,
...     split_by_node=True,
...     shuffle_shards=True,
... )
```

Return type *CutSet*

to_dicts()

Return type Iterable[dict]

decompose(*output_dir=None*, *verbose=False*)

Return a 3-tuple of unique (recordings, supervisions, features) found in this `CutSet`. Some manifest sets may also be `None`, e.g., if features were not extracted.

Note: `MixedCut` is iterated over its track cuts.

Parameters

- **output_dir** (`Union[Path, str, None]`) – directory where the manifests will be saved. The following files will be created: 'recordings.jsonl.gz', 'supervisions.jsonl.gz', 'features.jsonl.gz'.
- **verbose** (`bool`) – when `True`, shows a progress bar.

Return type `Tuple[Optional[RecordingSet], Optional[SupervisionSet], Optional[FeatureSet]]`

describe()

Print a message describing details about the `CutSet` - the number of cuts and the duration statistics, including the total duration and the percentage of speech segments.

Example output: Cuts count: 547 Total duration (hours): 326.4 Speech duration (hours): 79.6 (24.4%) *** Duration statistics (seconds): mean 2148.0 std 870.9 min 477.0 25% 1523.0 50% 2157.0 75% 2423.0 99% 2500.0 99.5% 2523.0 99.9% 2601.0 max 5415.0 dtype: float64

Return type `None`

split(*num_splits*, *shuffle=False*, *drop_last=False*)

Split the `CutSet` into `num_splits` pieces of equal size.

Parameters

- **num_splits** (`int`) – Requested number of splits.
- **shuffle** (`bool`) – Optionally shuffle the recordings order first.
- **drop_last** (`bool`) – determines how to handle splitting when `len(seq)` is not divisible by `num_splits`. When `False` (default), the splits might have unequal lengths. When `True`, it may discard the last element in some splits to ensure they are equally long.

Return type `List[CutSet]`

Returns A list of `CutSet` pieces.

split_lazy(*output_dir*, *chunk_size*, *prefix=''*)

Splits a manifest (either lazily or eagerly opened) into chunks, each with `chunk_size` items (except for the last one, typically).

In order to be memory efficient, this implementation saves each chunk to disk in a `.jsonl.gz` format as the input manifest is sampled.

Note: For lowest memory usage, use `load_manifest_lazy` to open the input manifest for this method.

Parameters

- **it** – any iterable of Lhotse manifests.

- **output_dir** (Union[Path, str]) – directory where the split manifests are saved. Each manifest is saved at: `{output_dir}/{prefix}.{split_idx}.jsonl.gz`
- **chunk_size** (int) – the number of items in each chunk.
- **prefix** (str) – the prefix of each manifest.

Return type List[*CutSet*]

Returns a list of lazily opened chunk manifests.

subset(*, *supervision_ids=None*, *cut_ids=None*, *first=None*, *last=None*)
Return a new `CutSet` according to the selected subset criterion. Only a single argument to `subset` is supported at this time.

Example:

```
>>> cuts = CutSet.from_yaml('path/to/cuts')
>>> train_set = cuts.subset(supervision_ids=train_ids)
>>> test_set = cuts.subset(supervision_ids=test_ids)
```

Parameters

- **supervision_ids** (Optional[Iterable[str]]) – List of supervision IDs to keep.
- **cut_ids** (Optional[Iterable[str]]) – List of cut IDs to keep. The returned *CutSet* preserves the order of *cut_ids*.
- **first** (Optional[int]) – int, the number of first cuts to keep.
- **last** (Optional[int]) – int, the number of last cuts to keep.

Return type *CutSet*

Returns a new `CutSet` with the subset results.

filter_supervisions(*predicate*)
Return a new CutSet with Cuts containing only *SupervisionSegments* satisfying *predicate*

Cuts without supervisions are preserved

Example:

```
>>> cuts = CutSet.from_yaml('path/to/cuts')
>>> at_least_five_second_supervisions = cuts.filter_supervisions(lambda s:␣
↪s.duration >= 5)
```

Parameters **predicate** (Callable[[*SupervisionSegment*], bool]) – A callable that accepts *SupervisionSegment* and returns bool

Return type *CutSet*

Returns a CutSet with filtered supervisions

merge_supervisions(*custom_merge_fn=None*)
Return a copy of the cut that has all of its supervisions merged into a single segment.

The new start is the start of the earliest superivion, and the new duration is a minimum spanning duration for all the supervisions.

The text fields are concatenated with a whitespace, and all other string fields (including IDs) are prefixed with "cat#" and concatenated with a hash symbol "#". This is also applied to `custom` fields. Fields with a `None` value are omitted.

Parameters **custom_merge_fn** (`Optional[Callable[[str, Iterable[Any]], Any]]`) – a function that will be called to merge custom fields values. We expect `custom_merge_fn` to handle all possible custom keys. When not provided, we will treat all custom values as strings. It will be called roughly like: `custom_merge_fn(custom_key, [s.custom[custom_key] for s in sups])`

Return type *CutSet*

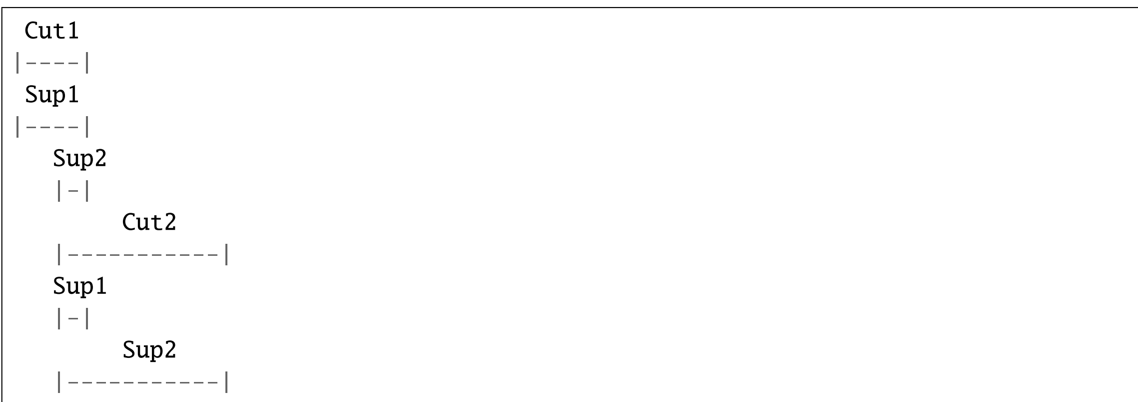
trim_to_supervisions(*keep_overlapping=True*, *min_duration=None*, *context_direction='center'*, *num_jobs=1*)

Return a new CutSet with Cuts that have identical spans as their supervisions.

For example, the following cut:

```
        Cut
|-----------------|
 Sup1
|----|  Sup2
   |-----------|
```

is transformed into two cuts:



```
 Cut1
|----|
 Sup1
|----|
   Sup2
   |-|
        Cut2
   |-----------|
   Sup1
   |-|
        Sup2
   |-----------|
```

Parameters

- **keep_overlapping** (`bool`) – when `False`, it will discard parts of other supervisions that overlap with the main supervision. In the illustration above, it would discard `Sup2` in `Cut1` and `Sup1` in `Cut2`. In this mode, we guarantee that there will always be exactly one supervision per cut.
- **min_duration** (`Optional[float]`) – An optional duration in seconds; specifying this argument will extend the cuts that would have been shorter than `min_duration` with actual acoustic context in the recording/features. If there are supervisions present in the context, they are kept when `keep_overlapping` is true. If there is not enough context, the returned cut will be shorter than `min_duration`. If the supervision segment is longer than `min_duration`, the return cut will be longer.
- **context_direction** (`Literal['center', 'left', 'right', 'random']`) – Which direction should the cut be expanded towards to include context. The value of "center" implies equal expansion to left and right; random uniformly samples a value between "left" and "right".
- **num_jobs** (`int`) – Number of parallel workers to process the cuts.

Return type *CutSet*

Returns a CutSet.

trim_to_unsupervised_segments()
Return a new CutSet with Cuts created from segments that have no supervisions (likely silence or noise).

Return type *CutSet*

Returns a CutSet.

mix_same_recording_channels()
Find cuts that come from the same recording and have matching start and end times, but represent different channels. Then, mix them together (in matching groups) and return a new CutSet that contains their mixes. This is useful for processing microphone array recordings.

It is intended to be used as the first operation after creating a new CutSet (but might also work in other circumstances, e.g. if it was cut to windows first).

Example:

```
>>> ami = prepare_ami('path/to/ami')
>>> cut_set = CutSet.from_manifests(recordings=ami['train']['recordings'])
>>> multi_channel_cut_set = cut_set.mix_same_recording_channels()
```

In the AMI example, the multi_channel_cut_set will yield MixedCuts that hold all single-channel Cuts together.

Return type *CutSet*

sort_by_duration(*ascending=False*)
Sort the CutSet according to cuts duration and return the result. Descending by default.

Return type *CutSet*

sort_like(*other*)
Sort the CutSet according to the order of cut IDs in other and return the result.

Return type *CutSet*

index_supervisions(*index_mixed_tracks=False*, *keep_ids=None*)
Create a two-level index of supervision segments. It is a mapping from a Cut's ID to an interval tree that contains the supervisions of that Cut.

The interval tree can be efficiently queried for overlapping and/or enveloping segments. It helps speed up some operations on Cuts of very long recordings (1h+) that contain many supervisions.

Parameters

- **index_mixed_tracks** (bool) – Should the tracks of MixedCut's be indexed as additional, separate entries.
- **keep_ids** (Optional[Set[str]]) – If specified, we will only index the supervisions with the specified IDs.

Return type Dict[str, IntervalTree]

Returns a mapping from MonoCut ID to an interval tree of SupervisionSegments.

pad(*duration=None*, *num_frames=None*, *num_samples=None*, *pad_feat_value=- 23.025850929940457*, *direction='right'*, *preserve_id=False*, *pad_value_dict=None*)
Return a new CutSet with Cuts padded to duration, num_frames or num_samples. Cuts longer than the specified argument will not be affected. By default, cuts will be padded to the right (i.e. after the signal).

When none of duration, num_frames, or num_samples is specified, we'll try to determine the best way to pad to the longest cut based on whether features or recordings are available.

Parameters

- **duration** (Optional[float]) – The cuts minimal duration after padding. When not specified, we'll choose the duration of the longest cut in the CutSet.
- **num_frames** (Optional[int]) – The cut's total number of frames after padding.
- **num_samples** (Optional[int]) – The cut's total number of samples after padding.
- **pad_feat_value** (float) – A float value that's used for padding the features. By default we assume a log-energy floor of approx. -23 (1e-10 after exp).
- **direction** (str) – string, 'left', 'right' or 'both'. Determines whether the padding is added before or after the cut.
- **preserve_id** (bool) – When True, preserves the cut ID from before padding. Otherwise, generates a new random ID (default).
- **pad_value_dict** (Optional[Dict[str, Union[int, float]]]) – Optional dict that specifies what value should be used for padding arrays in custom attributes.

Return type *CutSet*

Returns A padded CutSet.

truncate(*max_duration*, *offset_type*, *keep_excessive_supervisions=True*, *preserve_id=False*, *rng=None*)

Return a new CutSet with the Cuts truncated so that their durations are at most *max_duration*. Cuts shorter than *max_duration* will not be changed. :type max_duration: float :param max_duration: float, the maximum duration in seconds of a cut in the resulting manifest. :type offset_type: str :param offset_type: str, can be: - 'start' => cuts are truncated from their start; - 'end' => cuts are truncated from their end minus max_duration; - 'random' => cuts are truncated randomly between their start and their end minus max_duration :type keep_excessive_supervisions: bool :param keep_excessive_supervisions: bool. When a cut is truncated in the middle of a supervision segment, should the supervision be kept. :type preserve_id: bool :param preserve_id: bool. Should the truncated cut keep the same ID or get a new, random one. :type rng: Optional[Random] :param rng: optional random number generator to be used with a 'random' offset_type. :rtype: *CutSet* :return: a new CutSet instance with truncated cuts.

extend_by(*duration*, *direction='both'*, *preserve_id=False*)

Returns a new CutSet with cuts extended by *duration* amount.

Parameters

- **duration** (float) – float (seconds), specifies the duration by which the CutSet is extended.
- **direction** (str) – string, 'left', 'right' or 'both'. Determines whether to extend on the left, right, or both sides. If 'both', extend on both sides by the same duration (equal to *duration*).
- **preserve_id** (bool) – bool. Should the extended cut keep the same ID or get a new, random one.

Return type *CutSet*

Returns a new CutSet instance.

cut_into_windows(*duration*, *hop=None*, *keep_excessive_supervisions=True*, *num_jobs=1*)

Return a new CutSet, made by traversing each MonoCut in windows of duration seconds by hop seconds and creating new MonoCut out of them.

The last window might have a shorter duration if there was not enough audio, so you might want to use either `.filter()` or `.pad()` afterwards to obtain a uniform duration `CutSet`.

Parameters

- **duration** (float) – Desired duration of the new cuts in seconds.
- **hop** (Optional[float]) – Shift between the windows in the new cuts in seconds.
- **keep_excessive_supervisions** (bool) – bool. When a cut is truncated in the middle of a supervision segment, should the supervision be kept.
- **num_jobs** (int) – The number of parallel workers.

Return type *CutSet*

Returns a new CutSet with cuts made from shorter duration windows.

sample(*n_cuts=1*)

Randomly sample this `CutSet` and return `n_cuts` cuts. When `n_cuts` is 1, will return a single cut instance; otherwise will return a `CutSet`.

Return type Union[*Cut*, *CutSet*]

resample(*sampling_rate*, *affix_id=False*)

Return a new *CutSet* that contains cuts resampled to the new `sampling_rate`. All cuts in the manifest must contain recording information. If the feature manifests are attached, they are dropped.

Parameters

- **sampling_rate** (int) – The new sampling rate.
- **affix_id** (bool) – Should we modify the ID (useful if both versions of the same cut are going to be present in a single manifest).

Return type *CutSet*

Returns a modified copy of the `CutSet`.

perturb_speed(*factor*, *affix_id=True*)

Return a new *CutSet* that contains speed perturbed cuts with a factor of `factor`. It requires the recording manifests to be present. If the feature manifests are attached, they are dropped. The supervision manifests are modified to reflect the speed perturbed start times and durations.

Parameters

- **factor** (float) – The resulting playback speed is `factor` times the original one.
- **affix_id** (bool) – Should we modify the ID (useful if both versions of the same cut are going to be present in a single manifest).

Return type *CutSet*

Returns a modified copy of the `CutSet`.

perturb_tempo(*factor*, *affix_id=True*)

Return a new *CutSet* that contains tempo perturbed cuts with a factor of `factor`.

Compared to speed perturbation, tempo preserves pitch. It requires the recording manifests to be present. If the feature manifests are attached, they are dropped. The supervision manifests are modified to reflect the tempo perturbed start times and durations.

Parameters

- **factor** (float) – The resulting playback tempo is `factor` times the original one.

- **affix_id** (bool) – Should we modify the ID (useful if both versions of the same cut are going to be present in a single manifest).

Return type *CutSet*

Returns a modified copy of the CutSet.

perturb_volume(*factor*, *affix_id=True*)

Return a new *CutSet* that contains volume perturbed cuts with a factor of factor. It requires the recording manifests to be present. If the feature manifests are attached, they are dropped. The supervision manifests are remaining the same.

Parameters

- **factor** (float) – The resulting playback volume is factor times the original one.
- **affix_id** (bool) – Should we modify the ID (useful if both versions of the same cut are going to be present in a single manifest).

Return type *CutSet*

Returns a modified copy of the CutSet.

reverb_rir(*rir_recordings*, *normalize_output=True*, *early_only=False*, *affix_id=True*, *rir_channels=[0]*)

Return a new *CutSet* that contains original cuts convolved with randomly chosen impulse responses from *rir_recordings*. It requires the recording manifests to be present. If the feature manifests are attached, they are dropped. The supervision manifests remain the same.

Parameters

- **rir_recordings** (*RecordingSet*) – RecordingSet containing the room impulse responses.
- **normalize_output** (bool) – When true, output will be normalized to have energy as input.
- **early_only** (bool) – When true, only the early reflections (first 50 ms) will be used.
- **affix_id** (bool) – Should we modify the ID (useful if both versions of the same cut are going to be present in a single manifest).
- **rir_channels** (List[int]) – The channels of the impulse response to use. By default, first channel will be used. If it is a multi-channel RIR, applying RIR will produce MixedCut.

Return type *CutSet*

Returns a modified copy of the CutSet.

mix(*cuts*, *duration=None*, *allow_padding=False*, *snr=20*, *preserve_id=None*, *mix_prob=1.0*)

Mix cuts in this CutSet with randomly sampled cuts from another CutSet. A typical application would be data augmentation with noise, music, babble, etc.

Parameters

- **cuts** (*CutSet*) – a CutSet containing cuts to be mixed into this CutSet.
- **duration** (Optional[float]) – an optional float in seconds. When None, we will preserve the duration of the cuts in self (i.e. we'll truncate the mix if it exceeded the original duration). Otherwise, we will keep sampling cuts to mix in until we reach the specified duration (and truncate to that value, should it be exceeded).
- **allow_padding** (bool) – an optional bool. When it is True, we will allow the offset to be larger than the reference cut by padding the reference cut.

- **snr** (Union[float, Sequence[float], None]) – an optional float, or pair (range) of floats, in decibels. When it's a single float, we will mix all cuts with this SNR level (where cuts in self are treated as signals, and cuts in cuts are treated as noise). When it's a pair of floats, we will uniformly sample SNR values from that range. When None, we will mix the cuts without any level adjustment (could be too noisy for data augmentation).
- **preserve_id** (Optional[str]) – optional string ("left", "right"). when specified, append will preserve the cut id of the left- or right-hand side argument. otherwise, a new random id is generated.
- **mix_prob** (float) – an optional float in range [0, 1]. Specifies the probability of performing a mix. Values lower than 1.0 mean that some cuts in the output will be unchanged.

Return type CutSet

Returns a new CutSet with mixed cuts.

drop_features()

Return a new CutSet, where each Cut is copied and detached from its extracted features.

Return type CutSet

drop_recordings()

Return a new CutSet, where each Cut is copied and detached from its recordings.

Return type CutSet

drop_supervisions()

Return a new CutSet, where each Cut is copied and detached from its supervisions.

Return type CutSet

compute_and_store_features(*extractor*, *storage_path*, *num_jobs=None*, *augment_fn=None*, *storage_type=<class 'lhotse.features.io.LilcomChunkyWriter'>*, *executor=None*, *mix_eagerly=True*, *progress_bar=True*)

Extract features for all cuts, possibly in parallel, and store them using the specified storage object.

Examples:

Extract fbank features on one machine using 8 processes, store arrays partitioned in 8 archive files with lilcom compression:

```
>>> cuts = CutSet(...)
... cuts.compute_and_store_features(
...     extractor=Fbank(),
...     storage_path='feats',
...     num_jobs=8,
... )
```

Extract fbank features on one machine using 8 processes, store each array in a separate file with lilcom compression:

```
>>> cuts = CutSet(...)
... cuts.compute_and_store_features(
...     extractor=Fbank(),
...     storage_path='feats',
...     num_jobs=8,
...     storage_type=LilcomFilesWriter
... )
```

Extract fbank features on multiple machines using a Dask cluster with 80 jobs, store arrays partitioned in 80 archive files with lilcom compression:

```
>>> from distributed import Client
... cuts = CutSet(...)
... cuts.compute_and_store_features(
...     extractor=Fbank(),
...     storage_path='feats',
...     num_jobs=80,
...     executor=Client(...)
... )
```

Extract fbank features on one machine using 8 processes, store each array in an S3 bucket (requires `smart_open`):

```
>>> cuts = CutSet(...)
... cuts.compute_and_store_features(
...     extractor=Fbank(),
...     storage_path='s3://my-feature-bucket/my-corpus-features',
...     num_jobs=8,
...     storage_type=LilcomURLWriter
... )
```

Parameters

- **extractor** (*FeatureExtractor*) – A `FeatureExtractor` instance (either Lhotse's built-in or a custom implementation).
- **storage_path** (`Union[Path, str]`) – The path to location where we will store the features. The exact type and layout of stored files will be dictated by the `storage_type` argument.
- **num_jobs** (`Optional[int]`) – The number of parallel processes used to extract the features. We will internally split the CutSet into this many chunks and process each chunk in parallel.
- **augment_fn** (`Optional[Callable[[ndarray, int], ndarray]]`) – an optional callable used for audio augmentation. Be careful with the types of augmentations used: if they modify the start/end/duration times of the cut and its supervisions, you will end up with incorrect supervision information when using this API. E.g. for speed perturbation, use `CutSet.perturb_speed()` instead.
- **storage_type** (`Type[~FW]`) – a `FeaturesWriter` subclass type. It determines how the features are stored to disk, e.g. separate file per array, HDF5 files with multiple arrays, etc.
- **executor** (`Optional[Executor]`) – when provided, will be used to parallelize the feature extraction process. By default, we will instantiate a ProcessPoolExecutor. Learn more about the `Executor` API at https://lhotse.readthedocs.io/en/latest/parallelism.html
- **mix_eagerly** (`bool`) – Related to how the features are extracted for `MixedCut` instances, if any are present. When False, extract and store the features for each track separately, and mix them dynamically when loading the features. When True, mix the audio first and store the mixed features, returning a new `MonoCut` instance with the same ID. The returned `MonoCut` will not have a `Recording` attached.
- **progress_bar** (`bool`) – Should a progress bar be displayed (automatically turned off for parallel computation).

Return type *CutSet*

Returns Returns a new CutSet with Features manifests attached to the cuts.

compute_and_store_features_batch(*extractor*, *storage_path*, *manifest_path=None*, *batch_duration=600.0*, *num_workers=4*, *augment_fn=None*, *storage_type=<class 'lhotse.features.io.LilcomChunkyWriter'>*, *overwrite=False*)

Extract features for all cuts in batches. This method is intended for use with compatible feature extractors that implement an accelerated `extract_batch()` method. For example, `kaldifeat` extractors can be used this way (see, e.g., `KaldifeatFbank` or `KaldifeatMfcc`).

When a CUDA GPU is available and enabled for the feature extractor, this can be much faster than *CutSet.compute_and_store_features()*. Otherwise, the speed will be comparable to single-threaded extraction.

Example: extract fbank features on one GPU, using 4 dataloading workers for reading audio, and store the arrays in an archive file with lilcom compression:

```
>>> from lhotse import KaldifeatFbank, KaldifeatFbankConfig
>>> extractor = KaldifeatFbank(KaldifeatFbankConfig(device='cuda'))
>>> cuts = CutSet(...)
... cuts = cuts.compute_and_store_features_batch(
...     extractor=extractor,
...     storage_path='feats',
...     batch_duration=500,
...     num_workers=4,
... )
```

Parameters

- **extractor** (*FeatureExtractor*) – A *FeatureExtractor* instance, which should implement an accelerated `extract_batch` method.
- **storage_path** (Union[Path, str]) – The path to location where we will store the features. The exact type and layout of stored files will be dictated by the `storage_type` argument.
- **manifest_path** (Union[Path, str, None]) – Optional path where to write the CutSet manifest with attached feature manifests. If not specified, we will be keeping all manifests in memory.
- **batch_duration** (float) – The maximum number of audio seconds in a batch. Determines batch size dynamically.
- **num_workers** (int) – How many background dataloading workers should be used for reading the audio.
- **augment_fn** (Optional[Callable[[ndarray, int], ndarray]]) – an optional callable used for audio augmentation. Be careful with the types of augmentations used: if they modify the start/end/duration times of the cut and its supervisions, you will end up with incorrect supervision information when using this API. E.g. for speed perturbation, use `CutSet.perturb_speed()` instead.
- **storage_type** (Type[~FW]) – a `FeaturesWriter` subclass type. It determines how the features are stored to disk, e.g. separate file per array, HDF5 files with multiple arrays, etc.
- **overwrite** (bool) – should we overwrite the manifest, HDF5 files, etc. By default, this method will append to these files if they exist.

Return type *CutSet*

Returns Returns a new CutSet with Features manifests attached to the cuts.

compute_and_store_recordings(*storage_path*, *num_jobs=None*, *executor=None*, *augment_fn=None*, *progress_bar=True*)

Store waveforms of all cuts as audio recordings to disk.

Parameters

- **storage_path** (Union[Path, str]) – The path to location where we will store the audio recordings. For each cut, a sub-directory will be created that starts with the first 3 characters of the cut's ID. The audio recording is then stored in the sub-directory using the cut ID as filename and '.flac' as suffix.
- **num_jobs** (Optional[int]) – The number of parallel processes used to store the audio recordings. We will internally split the CutSet into this many chunks and process each chunk in parallel.
- **augment_fn** (Optional[Callable[[ndarray, int], ndarray]]) – an optional callable used for audio augmentation. Be careful with the types of augmentations used: if they modify the start/end/duration times of the cut and its supervisions, you will end up with incorrect supervision information when using this API. E.g. for speed perturbation, use `CutSet.perturb_speed()` instead.
- **executor** (Optional[Executor]) – when provided, will be used to parallelize the process. By default, we will instantiate a ProcessPoolExecutor. Learn more about the `Executor` API at https://lhotse.readthedocs.io/en/latest/parallelism.html
- **progress_bar** (bool) – Should a progress bar be displayed (automatically turned off for parallel computation).

Return type *CutSet*

Returns Returns a new `CutSet`.

save_audios(*storage_path*, *format='wav'*, *num_jobs=None*, *executor=None*, *augment_fn=None*, *progress_bar=True*)

Store waveforms of all cuts as audio recordings to disk.

Parameters

- **storage_path** (Union[Path, str]) – The path to location where we will store the audio recordings. For each cut, a sub-directory will be created that starts with the first 3 characters of the cut's ID. The audio recording is then stored in the sub-directory using filename `{cut.id}.{format}`
- **format** (str) – Audio format argument supported by `torchaudio.save`. Default is `wav`.
- **num_jobs** (Optional[int]) – The number of parallel processes used to store the audio recordings. We will internally split the CutSet into this many chunks and process each chunk in parallel.
- **augment_fn** (Optional[Callable[[ndarray, int], ndarray]]) – an optional callable used for audio augmentation. Be careful with the types of augmentations used: if they modify the start/end/duration times of the cut and its supervisions, you will end up with incorrect supervision information when using this API. E.g. for speed perturbation, use `CutSet.perturb_speed()` instead.
- **executor** (Optional[Executor]) – when provided, will be used to parallelize the process. By default, we will instantiate a ProcessPoolExecutor. Learn more about the `Executor` API at https://lhotse.readthedocs.io/en/latest/parallelism.html
- **progress_bar** (bool) – Should a progress bar be displayed (automatically turned off for parallel computation).

Return type *CutSet*

Returns Returns a new `CutSet`.

compute_global_feature_stats(*storage_path=None*, *max_cuts=None*)

Compute the global means and standard deviations for each feature bin in the manifest. It follows the implementation in scikit-learn: https://github.com/scikit-learn/scikit-learn/blob/0fb307bf39bbdacd6ed713c00724f8f871d60370/sklearn/utils/extmath.py#L715 which follows the paper: "Algorithms for computing the sample variance: analysis and recommendations", by Chan, Golub, and LeVeque.

Parameters

- **storage_path** (`Union[Path, str, None]`) – an optional path to a file where the stats will be stored with pickle.
- **max_cuts** (`Optional[int]`) – optionally, limit the number of cuts used for stats estimation. The cuts will be selected randomly in that case.

Return a dict of ``{'norm_means'``{'norm_means' np.ndarray, 'norm_stds': np.ndarray}``
with the shape of the arrays equal to the number of feature bins in this manifest.

Return type `Dict[str, ndarray]`

with_features_path_prefix(*path*)

Return type *CutSet*

with_recording_path_prefix(*path*)

Return type *CutSet*

copy_feats(*writer*, *output_path=None*)

Save a copy of every feature matrix found in this CutSet using `writer` and return a new manifest with cuts referring to the new feature locations.

Parameters

- **writer** (*FeaturesWriter*) – a *lhotse.features.io.FeaturesWriter* instance.
- **output_path** (`Union[Path, str, None]`) – optional path where the new manifest should be stored. It's used to write the manifest incrementally and return a lazy manifest, otherwise the copy is stored in memory.

Return type *CutSet*

Returns a copy of the manifest.

modify_ids(*transform_fn*)

Modify the IDs of cuts in this `CutSet`. Useful when combining multiple ``CutSet``s that were created from a single source, but contain features with different data augmentations techniques.

Parameters **transform_fn** (`Callable[[str], str]`) – A callable (function) that accepts a string (cut ID) and returns

a new string (new cut ID). :rtype: *CutSet* :return: a new `CutSet` with cuts with modified IDs.

fill_supervisions(*add_empty=True*, *shrink_ok=False*)

Fills the whole duration of each cut in a *CutSet* with a supervision segment.

If the cut has one supervision, its start is set to 0 and duration is set to `cut.duration`. Note: this may either expand a supervision that was shorter than a cut, or shrink a supervision that exceeds the cut.

If there are no supervisions, we will add an empty one when `add_empty==True`, otherwise we won't change anything.

If there are two or more supervisions, we will raise an exception.

Parameters

- **add_empty** (`bool`) – should we add an empty supervision with identical time bounds as the cut.
- **shrink_ok** (`bool`) – should we raise an error if a supervision would be shrank as a result of calling this method.

Return type *CutSet*

map_supervisions(*transform_fn*)

Modify the SupervisionSegments by *transform_fn* in this CutSet.

Parameters **transform_fn** (`Callable[[`*SupervisionSegment*`]`, *SupervisionSegment*`]`) – a function that modifies a supervision as an argument.

Return type *CutSet*

Returns a new, modified CutSet.

transform_text(*transform_fn*)

Return a copy of this `CutSet` with all `SupervisionSegments` text transformed with `transform_fn`. Useful for text normalization, phonetic transcription, etc.

Parameters **transform_fn** (`Callable[[str], str]`) – a function that accepts a string and returns a string.

Return type *CutSet*

Returns a new, modified CutSet.

filter(*predicate*)

Return a new manifest containing only the items that satisfy `predicate`. If the manifest is lazy, the filtering will also be applied lazily.

Parameters **predicate** (`Callable[[~T], bool]`) – a function that takes a cut as an argument and returns bool.

Returns a filtered manifest.

classmethod from_file(*path*)

Return type `Any`

classmethod from_json(*path*)

Return type `Any`

classmethod from_jsonl(*path*)

Return type `Any`

classmethod from_jsonl_lazy(*path*)

Read a JSONL manifest in a lazy manner, which opens the file but does not read it immediately. It is only suitable for sequential reads and iteration.

> **Warning:** Opening the manifest in this way might cause some methods that rely on random access to fail.

Return type `Any`

`classmethod from_yaml`(*path*)

Return type `Any`

`property is_lazy: bool`

Indicates whether this manifest was opened in lazy (read-on-the-fly) mode or not.

Return type `bool`

`map`(*transform_fn*)

Apply *transform_fn* to each item in this manifest and return a new manifest. If the manifest is opened lazy, the transform is also applied lazily.

Parameters **transform_fn** (`Callable`[[~T], ~T]) – A callable (function) that accepts a single item instance and returns a new (or the same) instance of the same type. E.g. with CutSet, callable accepts `Cut` and returns also `Cut`.

Returns a new `CutSet` with transformed cuts.

`classmethod mux`(**manifests*, *stop_early=False*, *weights=None*, *seed=0*)

Merges multiple manifest iterables into a new iterable by lazily multiplexing them during iteration time. If one of the iterables is exhausted before the others, we will keep iterating until all iterables are exhausted. This behavior can be changed with `stop_early` parameter.

Parameters

- **manifests** – iterables to be multiplexed. They can be either lazy or eager, but the resulting manifest will always be lazy.
- **stop_early** (`bool`) – should we stop the iteration as soon as we exhaust one of the manifests.
- **weights** (`Optional[List[Union[int, float]]]`) – an optional weight for each iterable, affects the probability of it being sampled. The weights are uniform by default. If lengths are known, it makes sense to pass them here for uniform distribution of items in the expectation.
- **seed** (`int`) – the random seed, ensures deterministic order across multiple iterations.

`classmethod open_writer`(*path*, *overwrite=True*)

Open a sequential writer that allows to store the manifests one by one, without the necessity of storing the whole manifest set in-memory. Supports writing to JSONL format (`.jsonl`), with optional gzip compression (`.jsonl.gz`).

> **Note:** when `path` is `None`, we will return a `InMemoryWriter` instead has the same API but stores the manifests in memory. It is convenient when you want to make disk saving optional.

Example:

```
>>> from lhotse import RecordingSet
... recordings = [...]
... with RecordingSet.open_writer('recordings.jsonl.gz') as writer:
```

(continues on next page)

(continued from previous page)

```
...     for recording in recordings:
...         writer.write(recording)
```

This writer can be useful for continuing to write files that were previously stopped – it will open the existing file and scan it for item IDs to skip writing them later. It can also be queried for existing IDs so that the user code may skip preparing the corresponding manifests.

Example:

```
>>> from lhotse import RecordingSet, Recording
... with RecordingSet.open_writer('recordings.jsonl.gz', overwrite=False) as␣
↪writer:
...     for path in Path('.').rglob('*.wav'):
...         recording_id = path.stem
...         if writer.contains(recording_id):
...             # Item already written previously - skip processing.
...             continue
...         # Item doesn't exist yet - run extra work to prepare the manifest
...         # and store it.
...         recording = Recording.from_file(path, recording_id=recording_id)
...         writer.write(recording)
```

Return type Union[SequentialJsonlWriter, InMemoryWriter]

repeat(*times=None*, *preserve_id=False*)

Return a new, lazily evaluated manifest that iterates over the original elements `times` number of times.

Parameters

- **times** (Optional[int]) – how many times to repeat (infinite by default).
- **preserve_id** (bool) – when True, we won't update the element ID with repeat number.

Returns a repeated manifest.

shuffle(*rng=None*, *buffer_size=10000*)

Shuffles the elements and returns a shuffled variant of self. If the manifest is opened lazily, performs shuffling on-the-fly with a fixed buffer size.

Parameters **rng** (Optional[Random]) – an optional instance of `random.Random` for precise control of randomness.

Returns a shuffled copy of self, or a manifest that is shuffled lazily.

to_eager()

Evaluates all lazy operations on this manifest, if any, and returns a copy that keeps all items in memory. If the manifest was "eager" already, this is a no-op and won't copy anything.

to_file(*path*)

Return type None

to_json(*path*)

Return type None

`to_jsonl`(*path*)

Return type None

`to_yaml`(*path*)

Return type None

`lhotse.cut.make_windowed_cuts_from_features`(*feature_set*, *cut_duration*, *cut_shift=None*, *keep_shorter_windows=False*)

Converts a FeatureSet to a CutSet by traversing each Features object in - possibly overlapping - windows, and creating a MonoCut out of that area. By default, the last window in traversal will be discarded if it cannot satisfy the *cut_duration* requirement.

Parameters

- **feature_set** (`FeatureSet`) – a FeatureSet object.
- **cut_duration** (`float`) – float, duration of created Cuts in seconds.
- **cut_shift** (`Optional[float]`) – optional float, specifies how many seconds are in between the starts of consecutive windows. Equals *cut_duration* by default.
- **keep_shorter_windows** (`bool`) – bool, when True, the last window will be used to create a MonoCut even if its duration is shorter than *cut_duration*.

Return type `CutSet`

Returns a CutSet object.

`lhotse.cut.mix`(*reference_cut*, *mixed_in_cut*, *offset=0*, *allow_padding=False*, *snr=None*, *preserve_id=None*)

Overlay, or mix, two cuts. Optionally the `mixed_in_cut` may be shifted by `offset` seconds and scaled down (positive SNR) or scaled up (negative SNR). Returns a MixedCut, which contains both cuts and the mix information. The actual feature mixing is performed during the call to `load_features()`.

Parameters

- **reference_cut** (`Cut`) – The reference cut for the mix - offset and snr are specified w.r.t this cut.
- **mixed_in_cut** (`Cut`) – The mixed-in cut - it will be offset and rescaled to match the offset and snr parameters.
- **offset** (`float`) – How many seconds to shift the `mixed_in_cut` w.r.t. the `reference_cut`.
- **allow_padding** (`bool`) – If the offset is larger than the cut duration, allow the cut to be padded.
- **snr** (`Optional[float]`) – Desired SNR of the `right_cut` w.r.t. the `left_cut` in the mix.
- **preserve_id** (`Optional[str]`) – optional string ("left", "right"). when specified, append will preserve the cut id of the left- or right-hand side argument. otherwise, a new random id is generated.

Return type `MixedCut`

Returns A `MixedCut` instance.

`lhotse.cut.pad`(*cut*, *duration=None*, *num_frames=None*, *num_samples=None*, *pad_feat_value=-23.025850929940457*, *direction='right'*, *preserve_id=False*, *pad_value_dict=None*)

Return a new MixedCut, padded with zeros in the recording, and `pad_feat_value` in each feature bin.

The user can choose to pad either to a specific *duration*; a specific number of frames *max_frames*; or a specific number of samples *num_samples*. The three arguments are mutually exclusive.

Parameters

- **cut** (*Cut*) – MonoCut to be padded.
- **duration** (Optional[float]) – The cut's minimal duration after padding.
- **num_frames** (Optional[int]) – The cut's total number of frames after padding.
- **num_samples** (Optional[int]) – The cut's total number of samples after padding.
- **pad_feat_value** (float) – A float value that's used for padding the features. By default we assume a log-energy floor of approx. -23 (1e-10 after exp).
- **direction** (str) – string, 'left', 'right' or 'both'. Determines whether the padding is added before or after the cut.
- **preserve_id** (bool) – When True, preserves the cut ID before padding. Otherwise, a new random ID is generated for the padded cut (default).
- **pad_value_dict** (Optional[Dict[str, Union[int, float]]]) – Optional dict that specifies what value should be used for padding arrays in custom attributes.

Return type *Cut*

Returns a padded MixedCut if duration is greater than this cut's duration, otherwise self.

lhotse.cut.**append**(*left_cut*, *right_cut*, *snr=None*, *preserve_id=None*)

Helper method for functional-style appending of Cuts.

Return type *MixedCut*

lhotse.cut.**mix_cuts**(*cuts*)

Return a MixedCut that consists of the input Cuts mixed with each other as-is.

Return type *MixedCut*

lhotse.cut.**append_cuts**(*cuts*)

Return a MixedCut that consists of the input Cuts appended to each other as-is.

Return type *Cut*

lhotse.cut.**compute_supervisions_frame_mask**(*cut*, *frame_shift=None*, *use_alignment_if_exists=None*)

Compute a mask that indicates which frames in a cut are covered by supervisions.

Parameters

- **cut** (*Cut*) – a cut object.
- **frame_shift** (Optional[float]) – optional frame shift in seconds; required when the cut does not have pre-computed features, otherwise ignored.
- **use_alignment_if_exists** (Optional[str]) – optional str (key from alignment dict); use the specified alignment type for generating the mask

:returns a 1D numpy array with value 1 for **frames** covered by at least one supervision, and 0 for **frames** not covered by any supervision.

lhotse.cut.**create_cut_set_eager**(*recordings=None*, *supervisions=None*, *features=None*, *output_path=None*, *random_ids=False*)

Create a *CutSet* from any combination of supervision, feature and recording manifests. At least one of recordings or features is required.

The created cuts will be of type *MonoCut*, even when the recordings have multiple channels. The *MonoCut* boundaries correspond to those found in the features, when available, otherwise to those found in the recordings.

When supervisions are provided, we'll be searching them for matching recording IDs and attaching to created cuts, assuming they are fully within the cut's time span.

Parameters

- **recordings** (Optional[*RecordingSet*]) – an optional *RecordingSet* manifest.
- **supervisions** (Optional[*SupervisionSet*]) – an optional *SupervisionSet* manifest.
- **features** (Optional[*FeatureSet*]) – an optional *FeatureSet* manifest.
- **output_path** (Union[Path, str, None]) – an optional path where the *CutSet* is stored.
- **random_ids** (bool) – boolean, should the cut IDs be randomized. By default, use the recording ID with a loop index and a channel idx, i.e. "{recording_id}-{idx}-{channel}")

Return type *CutSet*

Returns a new *CutSet* instance.

lhotse.cut.**create_cut_set_lazy**(*output_path*, *recordings=None*, *supervisions=None*, *features=None*, *random_ids=False*)

Create a *CutSet* from any combination of supervision, feature and recording manifests. At least one of recordings or features is required.

This method is the "lazy" variant, which allows to create a *CutSet* with a minimal memory usage. It has some extra requirements:

- **The user must provide an output_path, where we will write the cuts as** we create them. We'll return a lazily-opened *CutSet* from that file.
- **recordings and features (if both provided) have to be of equal length** and sorted by recording_id attribute of their elements.
- **supervisions (if provided) have to be sorted by recording_id;** note that there may be multiple supervisions with the same recording_id, which is allowed.

In addition, to prepare cuts in a fully memory-efficient way, make sure that:

- **All input manifests are stored in JSONL format and opened lazily** with <manifest_class>.from_jsonl_lazy(path) method.

For more details, see *create_cut_set_eager()*.

Parameters

- **output_path** (Union[Path, str]) – path to which we will write the cuts.
- **recordings** (Optional[*RecordingSet*]) – an optional *RecordingSet* manifest.
- **supervisions** (Optional[*SupervisionSet*]) – an optional *SupervisionSet* manifest.
- **features** (Optional[*FeatureSet*]) – an optional *FeatureSet* manifest.
- **random_ids** (bool) – boolean, should the cut IDs be randomized. By default, use the recording ID with a loop index and a channel idx, i.e. "{recording_id}-{idx}-{channel}")

Return type *CutSet*

Returns a new *CutSet* instance.

`lhotse.cut.merge_supervisions(cut, custom_merge_fn=None)`

Return a copy of the cut that has all of its supervisions merged into a single segment.

The new start is the start of the earliest superivion, and the new duration is a minimum spanning duration for all the supervisions.

The text fields are concatenated with a whitespace, and all other string fields (including IDs) are prefixed with "cat#" and concatenated with a hash symbol "#". This is also applied to `custom` fields. Fields with a `None` value are omitted.

Note: If you're using individual tracks of a *MixedCut*, note that this transform drops all the supervisions in individual tracks and assigns the merged supervision in the first *MonoCut* found in `self.tracks`.

Parameters **custom_merge_fn** (`Optional[Callable[[str, Iterable[Any]], Any]]`) – a function that will be called to merge custom fields values. We expect `custom_merge_fn` to handle all possible custom keys. When not provided, we will treat all custom values as strings. It will be called roughly like: `custom_merge_fn(custom_key, [s.custom[custom_key] for s in sups])`

Return type *Cut*

9.6 Recipes

Convenience methods used to prepare recording and supervision manifests for standard corpora.

9.7 Kaldi conversion

Convenience methods used to interact with Kaldi data directories.

`lhotse.kaldi.get_duration(path)`

Read a audio file, it supports pipeline style wave path and real waveform.

Parameters **path** (`Union[Path, str]`) – Path to an audio file or a Kaldi-style pipe.

Return type `float`

Returns float duration of the recording, in seconds.

`lhotse.kaldi.load_kaldi_data_dir(path, sampling_rate, frame_shift=None, map_string_to_underscores=None, num_jobs=1)`

Load a Kaldi data directory and convert it to a Lhotse RecordingSet and SupervisionSet manifests. For this to work, at least the wav.scp file must exist. SupervisionSet is created only when a segments file exists. All the other files (text, utt2spk, etc.) are optional, and some of them might not be handled yet. In particular, feats.scp files are ignored.

Parameters **map_string_to_underscores** (`Optional[str]`) – optional string, when specified, we will replace all instances of this string in SupervisonSegment IDs to underscores. This is to help with handling underscores in Kaldi (see *export_to_kaldi()*). This is also done for speaker IDs.

Return type `Tuple[`*RecordingSet*`, Optional[`*SupervisionSet*`], Optional[`*FeatureSet*`]]`

`lhotse.kaldi.`**`export_to_kaldi`**(*recordings*, *supervisions*, *output_dir*, *map_underscores_to=None*, *prefix_spk_id=False*)

Export a pair of `RecordingSet` and `SupervisionSet` to a Kaldi data directory. It even supports recordings that have multiple channels but the recordings will still have to have a single `AudioSource`.

The `RecordingSet` and `SupervisionSet` must be compatible, i.e. it must be possible to create a `CutSet` out of them.

Parameters

- **`recordings`** (*`RecordingSet`*) – a `RecordingSet` manifest.
- **`supervisions`** (*`SupervisionSet`*) – a `SupervisionSet` manifest.
- **`output_dir`** (`Union`[`Path`, `str`]) – path where the Kaldi-style data directory will be created.
- **`map_underscores_to`** (`Optional`[`str`]) – optional string with which we will replace all underscores. This helps avoid issues with Kaldi data dir sorting.
- **`prefix_spk_id`** (`Optional`[`bool`]) – add speaker_id as a prefix of utterance_id (this is to ensure correct sorting inside files which is required by Kaldi)

`lhotse.kaldi.`**`load_kaldi_text_mapping`**(*path*, *must_exist=False*)

Load Kaldi files such as utt2spk, spk2gender, text, etc. as a dict.

Return type `Dict`[`str`, `Optional`[`str`]]

`lhotse.kaldi.`**`save_kaldi_text_mapping`**(*data*, *path*)

Save flat dicts to Kaldi files such as utt2spk, spk2gender, text, etc.

`lhotse.kaldi.`**`make_wavscp_channel_string_map`**(*source*, *sampling_rate*)

Return type `Dict`[`int`, `str`]

9.8 Others

Helper methods used throughout the codebase.

`lhotse.manipulation.`**`combine`**(**manifests*)

Combine multiple manifests of the same type into one.

Examples:

```
>>> # Pass several arguments
>>> combine(recording_set1, recording_set2, recording_set3)
>>> # Or pass a single list/tuple of manifests
>>> combine([supervision_set1, supervision_set2])
```

Return type ~Manifest

`lhotse.manipulation.`**`split_parallelize_combine`**(*num_jobs*, *manifest*, *fn*, **args*, ***kwargs*)

Convenience wrapper that parallelizes the execution of functions that transform manifests. It splits the manifests into `num_jobs` pieces, applies the function to each split, and then combines the splits.

This function is used internally in Lhotse to implement some parallel ops.

Example:

```
>>> from lhotse import CutSet, split_parallelize_combine
>>> cuts = CutSet(...)
>>> window_cuts = split_parallelize_combine(
...     16,
...     cuts,
...     CutSet.cut_into_windows,
...     duration=30.0
... )
```

Parameters

- **num_jobs** (int) – The number of parallel jobs.
- **manifest** (*~Manifest*) – The manifest to be processed.
- **fn** (Callable) – Function or method that transforms the manifest; the first parameter has to be manifest (for methods, they have to be methods on that manifests type, e.g. CutSet.cut_into_windows.
- **args** – positional arguments to fn.

:param kwargs keyword arguments to fn.

Return type ~Manifest

lhotse.manipulation.**to_manifest**(*items*)

Take an iterable of data types in Lhotse such as Recording, SupervisonSegment or Cut, and create the manifest of the corresponding type. When the iterable is empty, returns None.

Return type Optional[~Manifest]

CHAPTER

TEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

l

lhotse.audio, 103
lhotse.augmentation, 255
lhotse.cut, 255
lhotse.dataset.collation, 44
lhotse.dataset.cut_transforms, 39
lhotse.dataset.diarization, 31
lhotse.dataset.input_strategies, 35
lhotse.dataset.sampling, 35
lhotse.dataset.signal_transforms, 41
lhotse.dataset.speech_recognition, 33
lhotse.dataset.unsupervised, 32
lhotse.dataset.vad, 35
lhotse.features.base, 130
lhotse.features.fbank, 226
lhotse.features.io, 240
lhotse.features.librosa_fbank, 236
lhotse.features.mfcc, 229
lhotse.features.mixer, 254
lhotse.features.spectrogram, 233
lhotse.kaldi, 316
lhotse.manipulation, 317
lhotse.recipes, 316
lhotse.supervision, 119

INDEX

Symbols

__call__() (*lhotse.dataset.input_strategies.AudioSamples method*), 36
__call__() (*lhotse.dataset.input_strategies.BatchIO method*), 35
__call__() (*lhotse.dataset.input_strategies.OnTheFlyFeatures method*), 37
__call__() (*lhotse.dataset.input_strategies.PrecomputedFeatures method*), 36
__init__() (*lhotse.audio.AudioLoadingError method*), 118
__init__() (*lhotse.audio.AudioMixer method*), 116
__init__() (*lhotse.audio.AudioSource method*), 104
__init__() (*lhotse.audio.DurationMismatchError method*), 118
__init__() (*lhotse.audio.Recording method*), 108
__init__() (*lhotse.audio.RecordingSet method*), 109
__init__() (*lhotse.cut.CutSet method*), 296
__init__() (*lhotse.cut.MixTrack method*), 282
__init__() (*lhotse.cut.MixedCut method*), 288
__init__() (*lhotse.cut.MonoCut method*), 269
__init__() (*lhotse.cut.PaddingCut method*), 277
__init__() (*lhotse.dataset.collation.TokenCollater method*), 44
__init__() (*lhotse.dataset.cut_transforms.CutConcatenate method*), 39
__init__() (*lhotse.dataset.cut_transforms.CutMix method*), 39
__init__() (*lhotse.dataset.cut_transforms.ExtraPadding method*), 40
__init__() (*lhotse.dataset.cut_transforms.PerturbSpeed method*), 40
__init__() (*lhotse.dataset.cut_transforms.PerturbTempo method*), 40
__init__() (*lhotse.dataset.cut_transforms.PerturbVolume method*), 40
__init__() (*lhotse.dataset.cut_transforms.ReverbWithImpulseResponse method*), 40
__init__() (*lhotse.dataset.diarization.DiarizationDataset method*), 32
__init__() (*lhotse.dataset.input_strategies.AudioSamples method*), 37
__init__() (*lhotse.dataset.input_strategies.BatchIO method*), 35
__init__() (*lhotse.dataset.input_strategies.OnTheFlyFeatures method*), 38
__init__() (*lhotse.dataset.signal_transforms.DereverbWPE method*), 44
__init__() (*lhotse.dataset.signal_transforms.GlobalMVN method*), 41
__init__() (*lhotse.dataset.signal_transforms.RandomizedSmoothing method*), 43
__init__() (*lhotse.dataset.signal_transforms.SpecAugment method*), 41
__init__() (*lhotse.dataset.source_separation.DynamicallyMixedSourceSe method*), 34
__init__() (*lhotse.dataset.source_separation.PreMixedSourceSeparation method*), 35
__init__() (*lhotse.dataset.speech_recognition.K2SpeechRecognitionData method*), 33
__init__() (*lhotse.dataset.unsupervised.DynamicUnsupervisedDataset method*), 33
__init__() (*lhotse.dataset.unsupervised.UnsupervisedDataset method*), 32
__init__() (*lhotse.dataset.unsupervised.UnsupervisedWaveformDataset method*), 32
__init__() (*lhotse.dataset.vad.VadDataset method*), 35
__init__() (*lhotse.features.base.FeatureExtractor method*), 130
__init__() (*lhotse.features.base.FeatureSet method*), 137
__init__() (*lhotse.features.base.FeatureSetBuilder method*), 142
__init__() (*lhotse.features.base.Features method*), 137
__init__() (*lhotse.features.base.TorchaudioFeatureExtractor method*), 134
__init__() (*lhotse.features.fbank.TorchaudioFbank method*), 227
__init__() (*lhotse.features.fbank.TorchaudioFbankConfig method*), 226
__init__() (*lhotse.features.io.ChunkedLilcomHdf5Reader method*), 246
__init__() (*lhotse.features.io.ChunkedLilcomHdf5Writer method*), 247

`__init__()` (*lhotse.features.io.KaldiReader method*), 250
`__init__()` (*lhotse.features.io.KaldiWriter method*), 251
`__init__()` (*lhotse.features.io.LilcomChunkyReader method*), 248
`__init__()` (*lhotse.features.io.LilcomChunkyWriter method*), 248
`__init__()` (*lhotse.features.io.LilcomFilesReader method*), 242
`__init__()` (*lhotse.features.io.LilcomFilesWriter method*), 242
`__init__()` (*lhotse.features.io.LilcomHdf5Reader method*), 245
`__init__()` (*lhotse.features.io.LilcomHdf5Writer method*), 245
`__init__()` (*lhotse.features.io.LilcomURLReader method*), 249
`__init__()` (*lhotse.features.io.LilcomURLWriter method*), 250
`__init__()` (*lhotse.features.io.MemoryLilcomReader method*), 252
`__init__()` (*lhotse.features.io.MemoryLilcomWriter method*), 252
`__init__()` (*lhotse.features.io.MemoryRawReader method*), 253
`__init__()` (*lhotse.features.io.MemoryRawWriter method*), 253
`__init__()` (*lhotse.features.io.NumpyFilesReader method*), 243
`__init__()` (*lhotse.features.io.NumpyFilesWriter method*), 243
`__init__()` (*lhotse.features.io.NumpyHdf5Reader method*), 244
`__init__()` (*lhotse.features.io.NumpyHdf5Writer method*), 244
`__init__()` (*lhotse.features.kaldi.extractors.Fbank method*), 143
`__init__()` (*lhotse.features.kaldi.extractors.Mfcc method*), 143
`__init__()` (*lhotse.features.kaldi.layers.Wav2FFT method*), 158
`__init__()` (*lhotse.features.kaldi.layers.Wav2LogFilterBank method*), 198
`__init__()` (*lhotse.features.kaldi.layers.Wav2LogSpec method*), 185
`__init__()` (*lhotse.features.kaldi.layers.Wav2MFCC method*), 212
`__init__()` (*lhotse.features.kaldi.layers.Wav2Spec method*), 171
`__init__()` (*lhotse.features.kaldi.layers.Wav2Win method*), 144
`__init__()` (*lhotse.features.librosa_fbank.LibrosaFbank method*), 238
`__init__()` (*lhotse.features.librosa_fbank.LibrosaFbankConfig method*), 236
`__init__()` (*lhotse.features.mfcc.TorchaudioMfcc method*), 230
`__init__()` (*lhotse.features.mfcc.TorchaudioMfccConfig method*), 230
`__init__()` (*lhotse.features.mixer.FeatureMixer method*), 254
`__init__()` (*lhotse.features.spectrogram.Spectrogram method*), 234
`__init__()` (*lhotse.features.spectrogram.SpectrogramConfig method*), 233
`__init__()` (*lhotse.supervision.AlignmentItem method*), 119
`__init__()` (*lhotse.supervision.SupervisionSegment method*), 123
`__init__()` (*lhotse.supervision.SupervisionSet method*), 124
`--absolute-paths <absolute_paths>`
    `lhotse-prepare-callhome-egyptian command line option`, 80
    `lhotse-prepare-callhome-english command line option`, 81
    `lhotse-prepare-cmu-kids command line option`, 82
    `lhotse-prepare-cslu-kids command line option`, 83
    `lhotse-prepare-eval2000 command line option`, 85
    `lhotse-prepare-fisher-english command line option`, 85
    `lhotse-prepare-fisher-spanish command line option`, 86
    `lhotse-prepare-gale-arabic command line option`, 86
    `lhotse-prepare-gale-mandarin command line option`, 87
    `lhotse-prepare-switchboard command line option`, 96
`--annotations <annotations>`
    `lhotse-download-ami command line option`, 61
    `lhotse-prepare-ami command line option`, 77
`--audio`
    `lhotse-cut-export-to-webdataset command line option`, 54
`--audio <audio>`
    `lhotse-prepare-gale-arabic command line option`, 86
    `lhotse-prepare-gale-mandarin command line option`, 87
`--audio-dirs <audio_dirs>`
    `lhotse-prepare-fisher-english command`

line option, 85
--audio-format <audio_format>
lhotse-cut-export-to-webdataset command line option, 54
--batch-duration <batch_duration>
lhotse-feat-extract-cuts-batch command line option, 71
--buck-walter
lhotse-prepare-mgb2 command line option, 92
--context-direction <context_direction>
lhotse-cut-trim-to-supervisions command line option, 57
--custom
lhotse-cut-export-to-webdataset command line option, 54
--cut-duration <cut_duration>
lhotse-cut-windowed command line option, 59
--cut-shift <cut_shift>
lhotse-cut-windowed command line option, 59
--cutids <cutids>
lhotse-subset command line option, 100
--dataset-part <dataset_part>
lhotse-prepare-nsc command line option, 94
--dataset-parts <dataset_parts>
lhotse-prepare-librispeech command line option, 91
lhotse-prepare-wenet-speech command line option, 98
--dev <dev>
lhotse-prepare-dihard3 command line option, 83
--discard-overflowing-supervisions
lhotse-cut-truncate command line option, 58
--discard-overlapping
lhotse-cut-trim-to-supervisions command line option, 57
--discard-shorter-windows
lhotse-cut-windowed command line option, 59
--dont-read-data
lhotse-validate command line option, 100
lhotse-validate-pair command line option, 101
--duration <duration>
lhotse-cut-pad command line option, 55
--eval <eval>
lhotse-prepare-dihard3 command line option, 83
--fault-tolerant
lhotse-cut-export-to-webdataset command line option, 54
--feature-manifest <feature_manifest>
lhotse-cut-simple command line option, 57
lhotse-feat-extract command line option, 69
lhotse-feat-extract-cuts command line option, 70
lhotse-feat-extract-cuts-batch command line option, 71
--feature-type <feature_type>
lhotse-feat-write-default-config command line option, 72
--features
lhotse-cut-export-to-webdataset command line option, 54
--first <first>
lhotse-subset command line option, 100
--flac
lhotse-prepare-mls command line option, 93
--force-download
lhotse-download-ali-meeting command line option, 61
lhotse-download-libricss command line option, 64
lhotse-download-voxceleb1 command line option, 68
lhotse-download-voxceleb2 command line option, 68
--force-download <force_download>
lhotse-download-ami command line option, 61
lhotse-download-icsi command line option, 64
--frame-shift <frame_shift>
lhotse-kaldi-import command line option, 74
--full
lhotse-download-librispeech command line option, 65
--host <host>
lhotse-download-gigaspeech command line option, 63
--install-dir <install_dir>
lhotse-install-sph2pipe command line option, 73
--keep-overflowing-supervisions
lhotse-cut-truncate command line option, 58
--keep-overlapping
lhotse-cut-trim-to-supervisions command line option, 57
--keep-shorter-windows

lhotse-cut-windowed command line option, 59
--lang <lang>
lhotse-download-mtedx command line option, 66
lhotse-prepare-mtedx command line option, 93
--language <language>
lhotse-prepare-commonvoice command line option, 82
--last <last>
lhotse-subset command line option, 100
--lilcom-tick-power <lilcom_tick_power>
lhotse-feat-extract command line option, 69
--link-previous-utterance
lhotse-prepare-libritts command line option, 91
--map-string-to-underscores <map_string_to_underscores>
lhotse-kaldi-import command line option, 74
--map-underscores-to <map_underscores_to>
lhotse-kaldi-export command line option, 74
--max-duration <max_duration>
lhotse-cut-truncate command line option, 58
--max-jobs <max_jobs>
lhotse-copy-feats command line option, 52
--mer-thresh <mer_thresh>
lhotse-prepare-mgb2 command line option, 92
--mic <mic>
lhotse-download-ami command line option, 61
lhotse-download-icsi command line option, 64
lhotse-prepare-ali-meeting command line option, 76
lhotse-prepare-ami command line option, 77
lhotse-prepare-aspire command line option, 77
lhotse-prepare-icsi command line option, 89
--min-duration <min_duration>
lhotse-cut-trim-to-supervisions command line option, 57
--min-segment-seconds <min_segment_seconds>
lhotse-prepare-librimix command line option, 90
--mini
lhotse-download-librispeech command line option, 65
--no-audio
lhotse-cut-export-to-webdataset command line option, 54
--no-buck-walter
lhotse-prepare-mgb2 command line option, 92
--no-custom
lhotse-cut-export-to-webdataset command line option, 54
--no-features
lhotse-cut-export-to-webdataset command line option, 54
--no-normalize-text
lhotse-prepare-earnings21 command line option, 84
lhotse-prepare-earnings22 command line option, 84
lhotse-prepare-spgispeech command line option, 95
--no-pad
lhotse-split command line option, 99
--no-precomputed-mixtures
lhotse-prepare-librimix command line option, 90
--no-previous-utterance
lhotse-prepare-libritts command line option, 91
--no-text-cleaning
lhotse-prepare-mgb2 command line option, 92
--no-uem
lhotse-prepare-dihard3 command line option, 83
--no-vocals
lhotse-prepare-musan command line option, 94
--normalize-text
lhotse-prepare-earnings21 command line option, 84
lhotse-prepare-earnings22 command line option, 84
lhotse-prepare-icsi command line option, 89
lhotse-prepare-spgispeech command line option, 95
--normalize-text <normalize_text>
lhotse-prepare-ami command line option, 77
lhotse-prepare-cslu-kids command line option, 83
--num_jobs <num_jobs>
lhotse-prepare-bvcc command line option, 79

--num-jobs <num_jobs>
lhotse-feat-extract command line option, 69
lhotse-feat-extract-cuts command line option, 70
lhotse-feat-extract-cuts-batch command line option, 71
lhotse-feat-upload command line option, 71
lhotse-kaldi-import command line option, 74
lhotse-prepare-commonvoice command line option, 82
lhotse-prepare-dihard3 command line option, 83
lhotse-prepare-fisher-english command line option, 85
lhotse-prepare-gigaspeech command line option, 87
lhotse-prepare-hifitts command line option, 88
lhotse-prepare-librispeech command line option, 91
lhotse-prepare-libritts command line option, 91
lhotse-prepare-mgb2 command line option, 92
lhotse-prepare-mls command line option, 93
lhotse-prepare-mtedx command line option, 93
lhotse-prepare-spgispeech command line option, 95
lhotse-prepare-timit command line option, 97
lhotse-prepare-voxceleb command line option, 98
lhotse-prepare-wenet-speech command line option, 98

--num-phones <num_phones>
lhotse-prepare-timit command line option, 97

--offset-range <offset_range>
lhotse-cut-random-mixed command line option, 56

--offset-type <offset_type>
lhotse-cut-truncate command line option, 58

--omit-silence
lhotse-prepare-switchboard command line option, 96

--opus
lhotse-prepare-mls command line option, 93

--pad
lhotse-split command line option, 99

--partition <partition>
lhotse-prepare-ami command line option, 77

--parts <parts>
lhotse-prepare-rir-noise command line option, 95

--prefix-spk-id
lhotse-kaldi-export command line option, 74

--preserve-id
lhotse-cut-truncate command line option, 58

--read-data
lhotse-validate command line option, 100
lhotse-validate-pair command line option, 101

--recording-manifest <recording_manifest>
lhotse-cut-simple command line option, 57

--retain-silence
lhotse-prepare-switchboard command line option, 96

--root-dir <root_dir>
lhotse-feat-extract command line option, 69

--rttm-dir <rttm_dir>
lhotse-prepare-callhome-english command line option, 81

--sampling-rate <sampling_rate>
lhotse-prepare-librimix command line option, 90

--seed <seed>
lhotse command line option, 51

--segment-words <segment_words>
lhotse-prepare-gale-mandarin command line option, 87

--sentiment-dir <sentiment_dir>
lhotse-prepare-switchboard command line option, 96

--shard-size <shard_size>
lhotse-cut-export-to-webdataset command line option, 54

--shuffle
lhotse-split command line option, 99

--snr-range <snr_range>
lhotse-cut-random-mixed command line option, 56

--split <split>
lhotse-prepare-commonvoice command line option, 82

--stop-on-fail
lhotse-cut-export-to-webdataset command line option, 54

--storage-type <storage_type>
lhotse-copy-feats command line option, 52
lhotse-feat-extract command line option, 69
lhotse-feat-extract-cuts command line option, 70
lhotse-feat-extract-cuts-batch command line option, 71
--subset <subset>
lhotse-download-gigaspeech command line option, 63
lhotse-prepare-gigaspeech command line option, 87
--supervision-manifest <supervision_manifest>
lhotse-cut-simple command line option, 57
--text-cleaning
lhotse-prepare-mgb2 command line option, 92
--transcript <transcript>
lhotse-prepare-gale-arabic command line option, 86
lhotse-prepare-gale-mandarin command line option, 87
--transcript-dir <transcript_dir>
lhotse-prepare-callhome-english command line option, 81
lhotse-prepare-switchboard command line option, 96
--transcript-dirs <transcript_dirs>
lhotse-prepare-fisher-english command line option, 85
--transcripts-dir <transcripts_dir>
lhotse-download-icsi command line option, 64
lhotse-prepare-icsi command line option, 89
--type <type>
lhotse-prepare-libricss command line option, 90
--uem
lhotse-prepare-dihard3 command line option, 83
--url <url>
lhotse-download-ami command line option, 61
lhotse-download-icsi command line option, 64
lhotse-install-sph2pipe command line option, 73
--use-vocals
lhotse-prepare-musan command line option, 94
--voxceleb1 <voxceleb1>
lhotse-prepare-voxceleb command line option, 98
--voxceleb2 <voxceleb2>
lhotse-prepare-voxceleb command line option, 98
--with-precomputed-mixtures
lhotse-prepare-librimix command line option, 90
-ad
lhotse-prepare-fisher-english command line option, 85
-b
lhotse-feat-extract-cuts-batch command line option, 71
-c
lhotse-cut-trim-to-supervisions command line option, 57
-d
lhotse-cut-pad command line option, 55
lhotse-cut-trim-to-supervisions command line option, 57
lhotse-cut-truncate command line option, 58
lhotse-cut-windowed command line option, 59
-f
lhotse-cut-export-to-webdataset command line option, 54
lhotse-cut-simple command line option, 57
lhotse-feat-extract command line option, 69
lhotse-feat-extract-cuts command line option, 70
lhotse-feat-extract-cuts-batch command line option, 71
lhotse-feat-write-default-config command line option, 72
lhotse-kaldi-import command line option, 74
-j
lhotse-copy-feats command line option, 52
lhotse-feat-extract command line option, 69
lhotse-feat-extract-cuts command line option, 70
lhotse-feat-extract-cuts-batch command line option, 71
lhotse-feat-upload command line option, 71
lhotse-kaldi-import command line option, 74
lhotse-prepare-commonvoice command line option, 82
lhotse-prepare-dihard3 command line

option, 83
lhotse-prepare-fisher-english command line option, 85
lhotse-prepare-gigaspeech command line option, 87
lhotse-prepare-hifitts command line option, 88
lhotse-prepare-librispeech command line option, 91
lhotse-prepare-libritts command line option, 91
lhotse-prepare-mgb2 command line option, 92
lhotse-prepare-mls command line option, 93
lhotse-prepare-mtedx command line option, 93
lhotse-prepare-spgispeech command line option, 95
lhotse-prepare-timit command line option, 97
lhotse-prepare-voxceleb command line option, 98
lhotse-prepare-wenet-speech command line option, 98

-l
lhotse-download-mtedx command line option, 66
lhotse-prepare-commonvoice command line option, 82
lhotse-prepare-mtedx command line option, 93

-nj
lhotse-prepare-bvcc command line option, 79

-o
lhotse-cut-random-mixed command line option, 56
lhotse-cut-truncate command line option, 58

-p
lhotse-kaldi-export command line option, 74
lhotse-prepare-librispeech command line option, 91
lhotse-prepare-nsc command line option, 94
lhotse-prepare-rir-noise command line option, 95
lhotse-prepare-timit command line option, 97
lhotse-prepare-wenet-speech command line option, 98

-r
lhotse-cut-simple command line option, 57
lhotse-feat-extract command line option, 69

-s
lhotse command line option, 51
lhotse-cut-export-to-webdataset command line option, 54
lhotse-cut-random-mixed command line option, 56
lhotse-cut-simple command line option, 57
lhotse-cut-windowed command line option, 59
lhotse-prepare-commonvoice command line option, 82
lhotse-prepare-gale-arabic command line option, 86
lhotse-prepare-gale-mandarin command line option, 87
lhotse-split command line option, 99

-t
lhotse-copy-feats command line option, 52
lhotse-feat-extract command line option, 69
lhotse-prepare-gale-arabic command line option, 86
lhotse-prepare-gale-mandarin command line option, 87

-td
lhotse-prepare-fisher-english command line option, 85

-u
lhotse-kaldi-export command line option, 74
lhotse-kaldi-import command line option, 74

-v1
lhotse-prepare-voxceleb command line option, 98

-v2
lhotse-prepare-voxceleb command line option, 98

A

add_module() (*lhotse.features.kaldi.layers.Wav2FFT method*), 158
add_module() (*lhotse.features.kaldi.layers.Wav2LogFilterBank method*), 198
add_module() (*lhotse.features.kaldi.layers.Wav2LogSpec method*), 185
add_module() (*lhotse.features.kaldi.layers.Wav2MFCC method*), 212
add_module() (*lhotse.features.kaldi.layers.Wav2Spec method*), 171

add_module() (*lhotse.features.kaldi.layers.Wav2Win method*), 145
add_to_mix() (*lhotse.audio.AudioMixer method*), 116
add_to_mix() (*lhotse.features.mixer.FeatureMixer method*), 255
alignment (*lhotse.supervision.SupervisionSegment attribute*), 121
AlignmentItem (*class in lhotse.supervision*), 119
append() (*in module lhotse.cut*), 314
append() (*lhotse.cut.Cut method*), 259
append() (*lhotse.cut.MixedCut method*), 288
append() (*lhotse.cut.MonoCut method*), 269
append() (*lhotse.cut.PaddingCut method*), 277
append_cuts() (*in module lhotse.cut*), 314
apply() (*lhotse.features.kaldi.layers.Wav2FFT method*), 159
apply() (*lhotse.features.kaldi.layers.Wav2LogFilterBank method*), 199
apply() (*lhotse.features.kaldi.layers.Wav2LogSpec method*), 185
apply() (*lhotse.features.kaldi.layers.Wav2MFCC method*), 212
apply() (*lhotse.features.kaldi.layers.Wav2Spec method*), 172
apply() (*lhotse.features.kaldi.layers.Wav2Win method*), 145
args (*lhotse.audio.AudioLoadingError attribute*), 118
args (*lhotse.audio.DurationMismatchError attribute*), 118
assert_and_maybe_fix_num_samples() (*in module lhotse.audio*), 117
AUDIO_DIR
    lhotse-download-icsi command line option, 64
    lhotse-prepare-broadcast-news command line option, 78
    lhotse-prepare-callhome-egyptian command line option, 80
    lhotse-prepare-callhome-english command line option, 81
    lhotse-prepare-fisher-spanish command line option, 86
    lhotse-prepare-icsi command line option, 89
    lhotse-prepare-switchboard command line option, 96
audio_energy() (*in module lhotse.audio*), 116
AudioLoadingError, 118
AudioMixer (*class in lhotse.audio*), 115
audioread_info() (*in module lhotse.audio*), 117
audioread_load() (*in module lhotse.audio*), 117
AudioSamples (*class in lhotse.dataset.input_strategies*), 36
AudioSource (*class in lhotse.audio*), 103
available_storage_backends() (*in module lhotse.features.io*), 241
available_windows() (*in module lhotse.features.kaldi.layers*), 225

B

BatchIO (*class in lhotse.dataset.input_strategies*), 35
bfloat16() (*lhotse.features.kaldi.layers.Wav2FFT method*), 159
bfloat16() (*lhotse.features.kaldi.layers.Wav2LogFilterBank method*), 199
bfloat16() (*lhotse.features.kaldi.layers.Wav2LogSpec method*), 186
bfloat16() (*lhotse.features.kaldi.layers.Wav2MFCC method*), 213
bfloat16() (*lhotse.features.kaldi.layers.Wav2Spec method*), 172
bfloat16() (*lhotse.features.kaldi.layers.Wav2Win method*), 146
buffers() (*lhotse.features.kaldi.layers.Wav2FFT method*), 160
buffers() (*lhotse.features.kaldi.layers.Wav2LogFilterBank method*), 199
buffers() (*lhotse.features.kaldi.layers.Wav2LogSpec method*), 186
buffers() (*lhotse.features.kaldi.layers.Wav2MFCC method*), 213
buffers() (*lhotse.features.kaldi.layers.Wav2Spec method*), 172
buffers() (*lhotse.features.kaldi.layers.Wav2Win method*), 146

C

cepstral_lifter (*lhotse.features.mfcc.TorchaudioMfccConfig attribute*), 230
channel (*lhotse.cut.MonoCut attribute*), 263
channel (*lhotse.supervision.SupervisionSegment attribute*), 121
channel_ids (*lhotse.audio.Recording property*), 107
channels (*lhotse.audio.AudioSource attribute*), 103
channels (*lhotse.audio.LibsndfileCompatibleAudioInfo property*), 116
channels (*lhotse.features.base.Features attribute*), 136
children() (*lhotse.features.kaldi.layers.Wav2FFT method*), 160
children() (*lhotse.features.kaldi.layers.Wav2LogFilterBank method*), 200
children() (*lhotse.features.kaldi.layers.Wav2LogSpec method*), 186
children() (*lhotse.features.kaldi.layers.Wav2MFCC method*), 213
children() (*lhotse.features.kaldi.layers.Wav2Spec method*), 173

children() (*lhotse.features.kaldi.layers.Wav2Win method*), 146
CHUNK_SIZE
lhotse-split-lazy command line option, 100
CHUNK_SIZE (*lhotse.features.io.LilcomChunkyReader attribute*), 248
CHUNK_SIZE (*lhotse.features.io.LilcomChunkyWriter attribute*), 248
ChunkedLilcomHdf5Reader (*class in lhotse.features.io*), 246
ChunkedLilcomHdf5Writer (*class in lhotse.features.io*), 247
close() (*lhotse.features.io.ChunkedLilcomHdf5Writer method*), 247
close() (*lhotse.features.io.KaldiWriter method*), 251
close() (*lhotse.features.io.LilcomChunkyWriter method*), 249
close() (*lhotse.features.io.LilcomHdf5Writer method*), 246
close() (*lhotse.features.io.MemoryLilcomWriter method*), 252
close() (*lhotse.features.io.MemoryRawWriter method*), 253
close() (*lhotse.features.io.NumpyHdf5Writer method*), 245
close_cached_file_handles() (*in module lhotse.features.io*), 244
collate_audio() (*in module lhotse.dataset.collation*), 45
collate_custom_field() (*in module lhotse.dataset.collation*), 45
collate_features() (*in module lhotse.dataset.collation*), 44
collate_matrices() (*in module lhotse.dataset.collation*), 46
collate_multi_channel_audio() (*in module lhotse.dataset.collation*), 46
collate_multi_channel_features() (*in module lhotse.dataset.collation*), 46
collate_vectors() (*in module lhotse.dataset.collation*), 46
combine() (*in module lhotse.manipulation*), 317
compute_and_store_features (*lhotse.cut.Cut attribute*), 257
compute_and_store_features() (*lhotse.cut.CutSet method*), 305
compute_and_store_features() (*lhotse.cut.MixedCut method*), 286
compute_and_store_features() (*lhotse.cut.MonoCut method*), 264
compute_and_store_features() (*lhotse.cut.PaddingCut method*), 276
compute_and_store_features_batch() (*lhotse.cut.CutSet method*), 307
compute_and_store_recording() (*lhotse.cut.Cut method*), 261
compute_and_store_recording() (*lhotse.cut.MixedCut method*), 288
compute_and_store_recording() (*lhotse.cut.MonoCut method*), 269
compute_and_store_recording() (*lhotse.cut.PaddingCut method*), 277
compute_and_store_recordings() (*lhotse.cut.CutSet method*), 308
compute_energy() (*lhotse.features.base.FeatureExtractor static method*), 131
compute_energy() (*lhotse.features.base.TorchaudioFeatureExtractor static method*), 134
compute_energy() (*lhotse.features.fbank.TorchaudioFbank static method*), 227
compute_energy() (*lhotse.features.kaldi.extractors.Fbank static method*), 143
compute_energy() (*lhotse.features.librosa_fbank.LibrosaFbank static method*), 237
compute_energy() (*lhotse.features.mfcc.TorchaudioMfcc static method*), 230
compute_energy() (*lhotse.features.spectrogram.Spectrogram static method*), 234
compute_features() (*lhotse.cut.Cut method*), 259
compute_features() (*lhotse.cut.MixedCut method*), 289
compute_features() (*lhotse.cut.MonoCut method*), 269
compute_features() (*lhotse.cut.PaddingCut method*), 278
compute_global_feature_stats() (*lhotse.cut.CutSet method*), 309
compute_global_stats() (*in module lhotse.features.base*), 142
compute_global_stats() (*lhotse.features.base.FeatureSet method*), 139
compute_supervisions_frame_mask() (*in module lhotse.cut*), 314
config_type (*lhotse.features.base.FeatureExtractor attribute*), 130
config_type (*lhotse.features.base.TorchaudioFeatureExtractor attribute*), 134
config_type (*lhotse.features.fbank.TorchaudioFbank attribute*), 227
config_type (*lhotse.features.kaldi.extractors.Fbank attribute*), 143
config_type (*lhotse.features.kaldi.extractors.Mfcc attribute*), 143
config_type (*lhotse.features.librosa_fbank.LibrosaFbank attribute*), 237
config_type (*lhotse.features.mfcc.TorchaudioMfcc at-

tribute), 230
config_type (*lhotse.features.spectrogram.Spectrogram attribute*), 233
copy_feats() (*lhotse.cut.CutSet method*), 309
copy_feats() (*lhotse.features.base.Features method*), 137
copy_feats() (*lhotse.features.base.FeatureSet method*), 139
CORPUS_DIR
lhotse-prepare-adept command line option, 75
lhotse-prepare-aidatatang-200zh command line option, 75
lhotse-prepare-aishell command line option, 76
lhotse-prepare-aishell4 command line option, 76
lhotse-prepare-ali-meeting command line option, 76
lhotse-prepare-ami command line option, 77
lhotse-prepare-aspire command line option, 78
lhotse-prepare-babel command line option, 78
lhotse-prepare-bvcc command line option, 79
lhotse-prepare-cmu-arctic command line option, 81
lhotse-prepare-cmu-indic command line option, 81
lhotse-prepare-cmu-kids command line option, 82
lhotse-prepare-commonvoice command line option, 82
lhotse-prepare-cslu-kids command line option, 83
lhotse-prepare-earnings21 command line option, 84
lhotse-prepare-earnings22 command line option, 84
lhotse-prepare-eval2000 command line option, 85
lhotse-prepare-fisher-english command line option, 85
lhotse-prepare-gigaspeech command line option, 88
lhotse-prepare-hifitts command line option, 88
lhotse-prepare-l2-arctic command line option, 89
lhotse-prepare-libricss command line option, 90
lhotse-prepare-librispeech command line option, 91
lhotse-prepare-libritts command line option, 91
lhotse-prepare-ljspeech command line option, 92
lhotse-prepare-mgb2 command line option, 92
lhotse-prepare-mls command line option, 93
lhotse-prepare-mtedx command line option, 93
lhotse-prepare-musan command line option, 94
lhotse-prepare-nsc command line option, 94
lhotse-prepare-peoples-speech command line option, 94
lhotse-prepare-rir-noise command line option, 95
lhotse-prepare-spgispeech command line option, 95
lhotse-prepare-timit command line option, 97
lhotse-prepare-vctk command line option, 97
lhotse-prepare-wenet-speech command line option, 98
lhotse-prepare-yesno command line option, 99
count() (*lhotse.audio.LibsndfileCompatibleAudioInfo method*), 117
cpu() (*lhotse.features.kaldi.layers.Wav2FFT method*), 160
cpu() (*lhotse.features.kaldi.layers.Wav2LogFilterBank method*), 200
cpu() (*lhotse.features.kaldi.layers.Wav2LogSpec method*), 186
cpu() (*lhotse.features.kaldi.layers.Wav2MFCC method*), 213
cpu() (*lhotse.features.kaldi.layers.Wav2Spec method*), 173
cpu() (*lhotse.features.kaldi.layers.Wav2Win method*), 147
create_cut_set_eager() (*in module lhotse.cut*), 314
create_cut_set_lazy() (*in module lhotse.cut*), 315
create_default_feature_extractor() (*in module lhotse.features.base*), 133
create_frame_window() (*in module lhotse.features.kaldi.layers*), 225
create_mel_scale() (*in module lhotse.features.kaldi.layers*), 225
cuda() (*lhotse.features.kaldi.layers.Wav2FFT method*), 160
cuda() (*lhotse.features.kaldi.layers.Wav2LogFilterBank

method), 200
cuda() (*lhotse.features.kaldi.layers.Wav2LogSpec method*), 186
cuda() (*lhotse.features.kaldi.layers.Wav2MFCC method*), 214
cuda() (*lhotse.features.kaldi.layers.Wav2Spec method*), 173
cuda() (*lhotse.features.kaldi.layers.Wav2Win method*), 147
custom (*lhotse.cut.MonoCut attribute*), 263
custom (*lhotse.cut.PaddingCut attribute*), 274
custom (*lhotse.supervision.SupervisionSegment attribute*), 121
Cut (*class in lhotse.cut*), 255
cut (*lhotse.cut.MixTrack attribute*), 282
cut_into_windows() (*lhotse.cut.Cut method*), 260
cut_into_windows() (*lhotse.cut.CutSet method*), 302
cut_into_windows() (*lhotse.cut.MixedCut method*), 289
cut_into_windows() (*lhotse.cut.MonoCut method*), 269
cut_into_windows() (*lhotse.cut.PaddingCut method*), 278
CUT_MANIFEST
    lhotse-cut-pad command line option, 56
    lhotse-cut-truncate command line option, 58
CUT_MANIFESTS
    lhotse-cut-append command line option, 53
    lhotse-cut-mix-by-recording-id command line option, 55
    lhotse-cut-mix-sequential command line option, 55
CutConcatenate (*class in lhotse.dataset.cut_transforms*), 39
CutMix (*class in lhotse.dataset.cut_transforms*), 39
CUTS
    lhotse-cut-trim-to-supervisions command line option, 58
CUTSET
    lhotse-cut-decompose command line option, 53
    lhotse-cut-describe command line option, 54
    lhotse-cut-export-to-webdataset command line option, 54
    lhotse-feat-extract-cuts command line option, 70
    lhotse-feat-extract-cuts-batch command line option, 71
CutSet (*class in lhotse.cut*), 293

D

data (*lhotse.audio.RecordingSet property*), 109
data (*lhotse.cut.CutSet property*), 296
data (*lhotse.features.base.FeatureSet property*), 137
data (*lhotse.supervision.SupervisionSet property*), 124
DATA_DIR
    lhotse-kaldi-import command line option, 75
decompose() (*lhotse.cut.CutSet method*), 297
DereverbWPE (*class in lhotse.dataset.signal_transforms*), 43
describe() (*lhotse.cut.CutSet method*), 298
device (*lhotse.features.base.FeatureExtractor property*), 131
device (*lhotse.features.base.TorchaudioFeatureExtractor property*), 134
device (*lhotse.features.fbank.TorchaudioFbank property*), 227
device (*lhotse.features.librosa_fbank.LibrosaFbank property*), 238
device (*lhotse.features.mfcc.TorchaudioMfcc property*), 230
device (*lhotse.features.spectrogram.Spectrogram property*), 234
DiarizationDataset (*class in lhotse.dataset.diarization*), 31
dither (*lhotse.features.fbank.TorchaudioFbankConfig attribute*), 226
dither (*lhotse.features.kaldi.layers.Wav2FFT property*), 158
dither (*lhotse.features.kaldi.layers.Wav2LogFilterBank property*), 200
dither (*lhotse.features.kaldi.layers.Wav2LogSpec property*), 187
dither (*lhotse.features.kaldi.layers.Wav2MFCC property*), 214
dither (*lhotse.features.kaldi.layers.Wav2Spec property*), 173
dither (*lhotse.features.mfcc.TorchaudioMfccConfig attribute*), 229
dither (*lhotse.features.spectrogram.SpectrogramConfig attribute*), 233
double() (*lhotse.features.kaldi.layers.Wav2FFT method*), 160
double() (*lhotse.features.kaldi.layers.Wav2LogFilterBank method*), 200
double() (*lhotse.features.kaldi.layers.Wav2LogSpec method*), 187
double() (*lhotse.features.kaldi.layers.Wav2MFCC method*), 214
double() (*lhotse.features.kaldi.layers.Wav2Spec method*), 173
double() (*lhotse.features.kaldi.layers.Wav2Win method*), 147
drop_features (*lhotse.cut.Cut attribute*), 257
drop_features() (*lhotse.cut.CutSet method*), 305

drop_features() (*lhotse.cut.MixedCut method*), 286
drop_features() (*lhotse.cut.MonoCut method*), 264
drop_features() (*lhotse.cut.PaddingCut method*), 276
drop_recording (*lhotse.cut.Cut attribute*), 258
drop_recording() (*lhotse.cut.MixedCut method*), 286
drop_recording() (*lhotse.cut.MonoCut method*), 264
drop_recording() (*lhotse.cut.PaddingCut method*), 276
drop_recordings() (*lhotse.cut.CutSet method*), 305
drop_supervisions (*lhotse.cut.Cut attribute*), 258
drop_supervisions() (*lhotse.cut.CutSet method*), 305
drop_supervisions() (*lhotse.cut.MixedCut method*), 286
drop_supervisions() (*lhotse.cut.MonoCut method*), 264
drop_supervisions() (*lhotse.cut.PaddingCut method*), 276
dump_patches (*lhotse.features.kaldi.layers.Wav2FFT attribute*), 161
dump_patches (*lhotse.features.kaldi.layers.Wav2LogFilterBank attribute*), 201
dump_patches (*lhotse.features.kaldi.layers.Wav2LogSpec attribute*), 187
dump_patches (*lhotse.features.kaldi.layers.Wav2MFCC attribute*), 214
dump_patches (*lhotse.features.kaldi.layers.Wav2Spec attribute*), 174
dump_patches (*lhotse.features.kaldi.layers.Wav2Win attribute*), 147
duration (*lhotse.audio.LibsndfileCompatibleAudioInfo property*), 117
duration (*lhotse.audio.Recording attribute*), 105
duration (*lhotse.cut.Cut attribute*), 257
duration (*lhotse.cut.MixedCut property*), 282
duration (*lhotse.cut.MonoCut attribute*), 263
duration (*lhotse.cut.PaddingCut attribute*), 273
duration (*lhotse.features.base.Features attribute*), 136
duration (*lhotse.supervision.AlignmentItem attribute*), 119
duration (*lhotse.supervision.SupervisionSegment attribute*), 121
duration() (*lhotse.audio.RecordingSet method*), 112
DurationMismatchError, 118
DynamicallyMixedSourceSeparationDataset (*class in lhotse.dataset.source_separation*), 34
DynamicUnsupervisedDataset (*class in lhotse.dataset.unsupervised*), 32

E

end (*lhotse.cut.Cut property*), 258
end (*lhotse.cut.MixedCut property*), 289
end (*lhotse.cut.MonoCut property*), 270
end (*lhotse.cut.PaddingCut property*), 278
end (*lhotse.features.base.Features property*), 136
end (*lhotse.supervision.AlignmentItem property*), 119
end (*lhotse.supervision.SupervisionSegment property*), 121
energy_floor (*lhotse.features.fbank.TorchaudioFbankConfig attribute*), 226
energy_floor (*lhotse.features.mfcc.TorchaudioMfccConfig attribute*), 229
energy_floor (*lhotse.features.spectrogram.SpectrogramConfig attribute*), 233
eval() (*lhotse.features.kaldi.layers.Wav2FFT method*), 161
eval() (*lhotse.features.kaldi.layers.Wav2LogFilterBank method*), 201
eval() (*lhotse.features.kaldi.layers.Wav2LogSpec method*), 187
eval() (*lhotse.features.kaldi.layers.Wav2MFCC method*), 214
eval() (*lhotse.features.kaldi.layers.Wav2Spec method*), 174
eval() (*lhotse.features.kaldi.layers.Wav2Win method*), 147
export_to_kaldi() (*in module lhotse.kaldi*), 316
extend_by (*lhotse.cut.Cut attribute*), 258
extend_by() (*lhotse.cut.CutSet method*), 302
extend_by() (*lhotse.cut.MixedCut method*), 284
extend_by() (*lhotse.cut.MonoCut method*), 265
extend_by() (*lhotse.cut.PaddingCut method*), 274
extra_repr() (*lhotse.features.kaldi.layers.Wav2FFT method*), 161
extra_repr() (*lhotse.features.kaldi.layers.Wav2LogFilterBank method*), 201
extra_repr() (*lhotse.features.kaldi.layers.Wav2LogSpec method*), 187
extra_repr() (*lhotse.features.kaldi.layers.Wav2MFCC method*), 215
extra_repr() (*lhotse.features.kaldi.layers.Wav2Spec method*), 174
extra_repr() (*lhotse.features.kaldi.layers.Wav2Win method*), 148
extract() (*lhotse.features.base.FeatureExtractor method*), 130
extract() (*lhotse.features.base.TorchaudioFeatureExtractor method*), 133
extract() (*lhotse.features.fbank.TorchaudioFbank method*), 227
extract() (*lhotse.features.kaldi.extractors.Fbank method*), 143
extract() (*lhotse.features.kaldi.extractors.Mfcc method*), 144
extract() (*lhotse.features.librosa_fbank.LibrosaFbank method*), 237
extract() (*lhotse.features.mfcc.TorchaudioMfcc method*), 230
extract() (*lhotse.features.spectrogram.Spectrogram*

method), 234
extract_batch() (*lhotse.features.base.FeatureExtractor method*), 131
extract_batch() (*lhotse.features.base.TorchaudioFeatureExtractor method*), 134
extract_batch() (*lhotse.features.fbank.TorchaudioFbank method*), 227
extract_batch() (*lhotse.features.librosa_fbank.LibrosaFbank method*), 238
extract_batch() (*lhotse.features.mfcc.TorchaudioMfcc method*), 231
extract_batch() (*lhotse.features.spectrogram.Spectrogram method*), 234
extract_from_recording_and_store() (*lhotse.features.base.FeatureExtractor method*), 132
extract_from_recording_and_store() (*lhotse.features.base.TorchaudioFeatureExtractor method*), 134
extract_from_recording_and_store() (*lhotse.features.fbank.TorchaudioFbank method*), 228
extract_from_recording_and_store() (*lhotse.features.librosa_fbank.LibrosaFbank method*), 238
extract_from_recording_and_store() (*lhotse.features.mfcc.TorchaudioMfcc method*), 231
extract_from_recording_and_store() (*lhotse.features.spectrogram.Spectrogram method*), 234
extract_from_samples_and_store() (*lhotse.features.base.FeatureExtractor method*), 131
extract_from_samples_and_store() (*lhotse.features.base.TorchaudioFeatureExtractor method*), 135
extract_from_samples_and_store() (*lhotse.features.fbank.TorchaudioFbank method*), 228
extract_from_samples_and_store() (*lhotse.features.librosa_fbank.LibrosaFbank method*), 238
extract_from_samples_and_store() (*lhotse.features.mfcc.TorchaudioMfcc method*), 231
extract_from_samples_and_store() (*lhotse.features.spectrogram.Spectrogram method*), 235
ExtraPadding (*class in lhotse.dataset.cut_transforms*), 39

F

Fbank (*class in lhotse.features.kaldi.extractors*), 143
feat_value (*lhotse.cut.PaddingCut attribute*), 273
feature_dim() (*lhotse.features.base.FeatureExtractor method*), 131
feature_dim() (*lhotse.features.base.TorchaudioFeatureExtractor method*), 135
feature_dim() (*lhotse.features.fbank.TorchaudioFbank method*), 227
feature_dim() (*lhotse.features.kaldi.extractors.Fbank method*), 143
feature_dim() (*lhotse.features.kaldi.extractors.Mfcc method*), 144
feature_dim() (*lhotse.features.librosa_fbank.LibrosaFbank method*), 237
feature_dim() (*lhotse.features.mfcc.TorchaudioMfcc method*), 230
feature_dim() (*lhotse.features.spectrogram.Spectrogram method*), 233
FEATURE_MANIFEST
    lhotse-cut-random-mixed command line option, 56
    lhotse-cut-windowed command line option, 59
    lhotse-feat-upload command line option, 72
FeatureExtractor (*class in lhotse.features.base*), 130
FeatureMixer (*class in lhotse.features.mixer*), 254
Features (*class in lhotse.features.base*), 136
features (*lhotse.cut.MonoCut attribute*), 263
features_type (*lhotse.cut.Cut attribute*), 257
features_type (*lhotse.cut.MixedCut property*), 283
features_type (*lhotse.cut.MonoCut property*), 263
features_type (*lhotse.cut.PaddingCut attribute*), 281
FeatureSet (*class in lhotse.features.base*), 137
FeatureSetBuilder (*class in lhotse.features.base*), 142
FeaturesReader (*class in lhotse.features.io*), 241
FeaturesWriter (*class in lhotse.features.io*), 240
fft_size (*lhotse.features.librosa_fbank.LibrosaFbankConfig attribute*), 236
fill_supervision (*lhotse.cut.Cut attribute*), 258
fill_supervision() (*lhotse.cut.MixedCut method*), 287
fill_supervision() (*lhotse.cut.MonoCut method*), 264
fill_supervision() (*lhotse.cut.PaddingCut method*), 276
fill_supervisions() (*lhotse.cut.CutSet method*), 309
filter() (*lhotse.audio.RecordingSet method*), 113
filter() (*lhotse.cut.CutSet method*), 310
filter() (*lhotse.features.base.FeatureSet method*), 139
filter() (*lhotse.supervision.SupervisionSet method*), 127
filter_supervisions (*lhotse.cut.Cut attribute*), 258
filter_supervisions() (*lhotse.cut.CutSet method*), 299

filter_supervisions() (*lhotse.cut.MixedCut method*), 288
filter_supervisions() (*lhotse.cut.MonoCut method*), 268
filter_supervisions() (*lhotse.cut.PaddingCut method*), 277
find() (*lhotse.features.base.FeatureSet method*), 139
find() (*lhotse.supervision.SupervisionSet method*), 127
float() (*lhotse.features.kaldi.layers.Wav2FFT method*), 161
float() (*lhotse.features.kaldi.layers.Wav2LogFilterBank method*), 201
float() (*lhotse.features.kaldi.layers.Wav2LogSpec method*), 187
float() (*lhotse.features.kaldi.layers.Wav2MFCC method*), 215
float() (*lhotse.features.kaldi.layers.Wav2Spec method*), 174
float() (*lhotse.features.kaldi.layers.Wav2Win method*), 148
fmax (*lhotse.features.librosa_fbank.LibrosaFbankConfig attribute*), 236
fmin (*lhotse.features.librosa_fbank.LibrosaFbankConfig attribute*), 236
forward() (*lhotse.dataset.signal_transforms.DereverbWPE method*), 44
forward() (*lhotse.dataset.signal_transforms.GlobalMVN method*), 41
forward() (*lhotse.dataset.signal_transforms.RandomizedSmoothing method*), 43
forward() (*lhotse.dataset.signal_transforms.SpecAugment method*), 42
forward() (*lhotse.features.kaldi.layers.Wav2FFT method*), 158
forward() (*lhotse.features.kaldi.layers.Wav2LogFilterBank method*), 201
forward() (*lhotse.features.kaldi.layers.Wav2LogSpec method*), 188
forward() (*lhotse.features.kaldi.layers.Wav2MFCC method*), 215
forward() (*lhotse.features.kaldi.layers.Wav2Spec method*), 174
forward() (*lhotse.features.kaldi.layers.Wav2Win method*), 145
frame_length (*lhotse.features.fbank.TorchaudioFbankConfig attribute*), 226
frame_length (*lhotse.features.kaldi.layers.Wav2FFT property*), 158
frame_length (*lhotse.features.kaldi.layers.Wav2LogFilterBank property*), 201
frame_length (*lhotse.features.kaldi.layers.Wav2LogSpec property*), 188
frame_length (*lhotse.features.kaldi.layers.Wav2MFCC property*), 215
frame_length (*lhotse.features.kaldi.layers.Wav2Spec property*), 174
frame_length (*lhotse.features.mfcc.TorchaudioMfccConfig attribute*), 229
frame_length (*lhotse.features.spectrogram.SpectrogramConfig attribute*), 233
frame_shift (*lhotse.cut.Cut attribute*), 257
frame_shift (*lhotse.cut.MixedCut property*), 283
frame_shift (*lhotse.cut.MonoCut property*), 263
frame_shift (*lhotse.cut.PaddingCut attribute*), 274
frame_shift (*lhotse.features.base.FeatureExtractor property*), 131
frame_shift (*lhotse.features.base.Features attribute*), 136
frame_shift (*lhotse.features.base.TorchaudioFeatureExtractor property*), 133
frame_shift (*lhotse.features.fbank.TorchaudioFbank property*), 229
frame_shift (*lhotse.features.fbank.TorchaudioFbankConfig attribute*), 226
frame_shift (*lhotse.features.kaldi.extractors.Fbank property*), 143
frame_shift (*lhotse.features.kaldi.extractors.Mfcc property*), 144
frame_shift (*lhotse.features.kaldi.layers.Wav2FFT property*), 158
frame_shift (*lhotse.features.kaldi.layers.Wav2LogFilterBank property*), 202
frame_shift (*lhotse.features.kaldi.layers.Wav2LogSpec property*), 188
frame_shift (*lhotse.features.kaldi.layers.Wav2MFCC property*), 215
frame_shift (*lhotse.features.kaldi.layers.Wav2Spec property*), 175
frame_shift (*lhotse.features.librosa_fbank.LibrosaFbank property*), 237
frame_shift (*lhotse.features.mfcc.TorchaudioMfcc property*), 232
frame_shift (*lhotse.features.mfcc.TorchaudioMfccConfig attribute*), 229
frame_shift (*lhotse.features.spectrogram.Spectrogram property*), 235
frame_shift (*lhotse.features.spectrogram.SpectrogramConfig attribute*), 233
frames (*lhotse.audio.LibsndfileCompatibleAudioInfo property*), 116
from_bytes() (*lhotse.audio.Recording static method*), 106
from_cuts() (*lhotse.cut.CutSet static method*), 296
from_cuts() (*lhotse.dataset.signal_transforms.GlobalMVN class method*), 41
from_dict (*lhotse.cut.Cut attribute*), 257
from_dict() (*lhotse.audio.AudioSource static method*), 104

from_dict() (*lhotse.audio.Recording static method*), 108
from_dict() (*lhotse.cut.MixedCut static method*), 288
from_dict() (*lhotse.cut.MixTrack static method*), 282
from_dict() (*lhotse.cut.MonoCut static method*), 268
from_dict() (*lhotse.cut.PaddingCut static method*), 277
from_dict() (*lhotse.features.base.FeatureExtractor class method*), 133
from_dict() (*lhotse.features.base.Features static method*), 137
from_dict() (*lhotse.features.base.TorchaudioFeatureExtractor class method*), 135
from_dict() (*lhotse.features.fbank.TorchaudioFbank class method*), 229
from_dict() (*lhotse.features.fbank.TorchaudioFbankConfig static method*), 226
from_dict() (*lhotse.features.librosa_fbank.LibrosaFbank class method*), 239
from_dict() (*lhotse.features.librosa_fbank.LibrosaFbankConfig static method*), 236
from_dict() (*lhotse.features.mfcc.TorchaudioMfcc class method*), 232
from_dict() (*lhotse.features.mfcc.TorchaudioMfccConfig static method*), 230
from_dict() (*lhotse.features.spectrogram.Spectrogram class method*), 236
from_dict() (*lhotse.features.spectrogram.SpectrogramConfig static method*), 233
from_dict() (*lhotse.supervision.SupervisionSegment static method*), 123
from_dicts() (*lhotse.audio.RecordingSet static method*), 110
from_dicts() (*lhotse.cut.CutSet static method*), 297
from_dicts() (*lhotse.features.base.FeatureSet static method*), 137
from_dicts() (*lhotse.supervision.SupervisionSet static method*), 124
from_dir() (*lhotse.audio.RecordingSet static method*), 110
from_features() (*lhotse.features.base.FeatureSet static method*), 137
from_file() (*lhotse.audio.Recording static method*), 105
from_file() (*lhotse.audio.RecordingSet class method*), 113
from_file() (*lhotse.cut.CutSet class method*), 310
from_file() (*lhotse.dataset.signal_transforms.GlobalMVN class method*), 41
from_file() (*lhotse.features.base.FeatureSet class method*), 140
from_file() (*lhotse.supervision.SupervisionSet class method*), 127
from_items() (*lhotse.audio.RecordingSet static method*), 110
from_items() (*lhotse.cut.CutSet static method*), 296
from_items() (*lhotse.features.base.FeatureSet static method*), 137
from_items() (*lhotse.supervision.SupervisionSet static method*), 124
from_json() (*lhotse.audio.RecordingSet class method*), 113
from_json() (*lhotse.cut.CutSet class method*), 310
from_json() (*lhotse.features.base.FeatureSet class method*), 140
from_json() (*lhotse.supervision.SupervisionSet class method*), 127
from_jsonl() (*lhotse.audio.RecordingSet class method*), 113
from_jsonl() (*lhotse.cut.CutSet class method*), 310
from_jsonl() (*lhotse.features.base.FeatureSet class method*), 140
from_jsonl() (*lhotse.supervision.SupervisionSet class method*), 127
from_jsonl_lazy() (*lhotse.audio.RecordingSet class method*), 113
from_jsonl_lazy() (*lhotse.cut.CutSet class method*), 310
from_jsonl_lazy() (*lhotse.features.base.FeatureSet class method*), 140
from_jsonl_lazy() (*lhotse.supervision.SupervisionSet class method*), 128
from_manifests() (*lhotse.cut.CutSet static method*), 296
from_recordings() (*lhotse.audio.RecordingSet static method*), 110
from_rttm() (*lhotse.supervision.SupervisionSet static method*), 124
from_segments() (*lhotse.supervision.SupervisionSet static method*), 124
from_webdataset() (*lhotse.cut.CutSet static method*), 297
from_yaml() (*lhotse.audio.RecordingSet class method*), 113
from_yaml() (*lhotse.cut.CutSet class method*), 311
from_yaml() (*lhotse.features.base.FeatureExtractor class method*), 133
from_yaml() (*lhotse.features.base.FeatureSet class method*), 140
from_yaml() (*lhotse.features.base.TorchaudioFeatureExtractor class method*), 135
from_yaml() (*lhotse.features.fbank.TorchaudioFbank class method*), 229
from_yaml() (*lhotse.features.librosa_fbank.LibrosaFbank class method*), 239
from_yaml() (*lhotse.features.mfcc.TorchaudioMfcc class method*), 232
from_yaml() (*lhotse.features.spectrogram.Spectrogram class method*), 236

from_yaml() (*lhotse.supervision.SupervisionSet class method*), 128

G

gender (*lhotse.supervision.SupervisionSegment attribute*), 121
get_buffer() (*lhotse.features.kaldi.layers.Wav2FFT method*), 161
get_buffer() (*lhotse.features.kaldi.layers.Wav2LogFilterBank method*), 202
get_buffer() (*lhotse.features.kaldi.layers.Wav2LogSpec method*), 188
get_buffer() (*lhotse.features.kaldi.layers.Wav2MFCC method*), 215
get_buffer() (*lhotse.features.kaldi.layers.Wav2Spec method*), 175
get_buffer() (*lhotse.features.kaldi.layers.Wav2Win method*), 148
get_duration() (*in module lhotse.kaldi*), 316
get_extra_state() (*lhotse.features.kaldi.layers.Wav2FFT method*), 162
get_extra_state() (*lhotse.features.kaldi.layers.Wav2LogFilterBank method*), 202
get_extra_state() (*lhotse.features.kaldi.layers.Wav2LogSpec method*), 188
get_extra_state() (*lhotse.features.kaldi.layers.Wav2MFCC method*), 216
get_extra_state() (*lhotse.features.kaldi.layers.Wav2Spec method*), 175
get_extra_state() (*lhotse.features.kaldi.layers.Wav2Win method*), 148
get_extractor_type() (*in module lhotse.features.base*), 133
get_memory_writer() (*in module lhotse.features.io*), 252
get_parameter() (*lhotse.features.kaldi.layers.Wav2FFT method*), 162
get_parameter() (*lhotse.features.kaldi.layers.Wav2LogFilterBank method*), 202
get_parameter() (*lhotse.features.kaldi.layers.Wav2LogSpec method*), 189
get_parameter() (*lhotse.features.kaldi.layers.Wav2MFCC method*), 216
get_parameter() (*lhotse.features.kaldi.layers.Wav2Spec method*), 175
get_parameter() (*lhotse.features.kaldi.layers.Wav2Win method*), 148
get_reader() (*in module lhotse.features.io*), 242
get_submodule() (*lhotse.features.kaldi.layers.Wav2FFT method*), 162
get_submodule() (*lhotse.features.kaldi.layers.Wav2LogFilterBank method*), 202
get_submodule() (*lhotse.features.kaldi.layers.Wav2LogSpec method*), 189
get_submodule() (*lhotse.features.kaldi.layers.Wav2MFCC method*), 216
get_submodule() (*lhotse.features.kaldi.layers.Wav2Spec method*), 175
get_submodule() (*lhotse.features.kaldi.layers.Wav2Win method*), 149
get_writer() (*in module lhotse.features.io*), 242
GlobalMVN (*class in lhotse.dataset.signal_transforms*), 41

H

half() (*lhotse.features.kaldi.layers.Wav2FFT method*), 163
half() (*lhotse.features.kaldi.layers.Wav2LogFilterBank method*), 203
half() (*lhotse.features.kaldi.layers.Wav2LogSpec method*), 189
half() (*lhotse.features.kaldi.layers.Wav2MFCC method*), 217
half() (*lhotse.features.kaldi.layers.Wav2Spec method*), 176
half() (*lhotse.features.kaldi.layers.Wav2Win method*), 149
has_features (*lhotse.cut.Cut attribute*), 257
has_features (*lhotse.cut.MixedCut property*), 282
has_features (*lhotse.cut.MonoCut property*), 263
has_features (*lhotse.cut.PaddingCut property*), 274
has_recording (*lhotse.cut.Cut attribute*), 257
has_recording (*lhotse.cut.MixedCut property*), 282
has_recording (*lhotse.cut.MonoCut property*), 263
has_recording (*lhotse.cut.PaddingCut property*), 274
high_freq (*lhotse.features.fbank.TorchaudioFbankConfig attribute*), 226
high_freq (*lhotse.features.mfcc.TorchaudioMfccConfig attribute*), 230
hop_size (*lhotse.features.librosa_fbank.LibrosaFbankConfig attribute*), 236

I

id (*lhotse.audio.Recording attribute*), 105
id (*lhotse.cut.Cut attribute*), 257
id (*lhotse.cut.MixedCut attribute*), 282
id (*lhotse.cut.MonoCut attribute*), 262
id (*lhotse.cut.PaddingCut attribute*), 273
id (*lhotse.supervision.SupervisionSegment attribute*), 121
ids (*lhotse.audio.RecordingSet property*), 110
ids (*lhotse.cut.CutSet property*), 296
ids (*lhotse.supervision.SupervisionSet property*), 124
index() (*lhotse.audio.LibsndfileCompatibleAudioInfo method*), 117
index_supervisions() (*lhotse.cut.Cut method*), 260
index_supervisions() (*lhotse.cut.CutSet method*), 301

index_supervisions() (*lhotse.cut.MixedCut method*), 289
index_supervisions() (*lhotse.cut.MonoCut method*), 270
index_supervisions() (*lhotse.cut.PaddingCut method*), 278
info() (*in module lhotse.audio*), 117
INPUT_MANIFEST
    lhotse-copy command line option, 52
    lhotse-copy-feats command line option, 52
inverse() (*lhotse.dataset.collation.TokenCollater method*), 44
inverse() (*lhotse.dataset.signal_transforms.GlobalMVN method*), 41
is_lazy (*lhotse.audio.RecordingSet property*), 114
is_lazy (*lhotse.cut.CutSet property*), 311
is_lazy (*lhotse.features.base.FeatureSet property*), 140
is_lazy (*lhotse.supervision.SupervisionSet property*), 128

K

K2SpeechRecognitionDataset (*class in lhotse.dataset.speech_recognition*), 33
KaldiReader (*class in lhotse.features.io*), 250
KaldiWriter (*class in lhotse.features.io*), 251

L

language (*lhotse.supervision.SupervisionSegment attribute*), 121
lhotse command line option
    --seed <seed>, 51
    -s, 51
lhotse.audio
    module, 103
lhotse.augmentation
    module, 255
lhotse.cut
    module, 255
lhotse.dataset.collation
    module, 44
lhotse.dataset.cut_transforms
    module, 39
lhotse.dataset.diarization
    module, 31
lhotse.dataset.input_strategies
    module, 35
lhotse.dataset.sampling
    module, 35
lhotse.dataset.signal_transforms
    module, 41
lhotse.dataset.speech_recognition
    module, 33
lhotse.dataset.unsupervised
    module, 32
lhotse.dataset.vad
    module, 35
lhotse.features.base
    module, 130
lhotse.features.fbank
    module, 226
lhotse.features.io
    module, 240
lhotse.features.librosa_fbank
    module, 236
lhotse.features.mfcc
    module, 229
lhotse.features.mixer
    module, 254
lhotse.features.spectrogram
    module, 233
lhotse.kaldi
    module, 316
lhotse.manipulation
    module, 317
lhotse.recipes
    module, 316
lhotse.supervision
    module, 119
lhotse-combine command line option
    MANIFESTS, 51
    OUTPUT_MANIFEST, 51
lhotse-copy command line option
    INPUT_MANIFEST, 52
    OUTPUT_MANIFEST, 52
lhotse-copy-feats command line option
    --max-jobs <max_jobs>, 52
    --storage-type <storage_type>, 52
    -j, 52
    -t, 52
    INPUT_MANIFEST, 52
    OUTPUT_MANIFEST, 52
    STORAGE_PATH, 52
lhotse-cut-append command line option
    CUT_MANIFESTS, 53
    OUTPUT_CUT_MANIFEST, 53
lhotse-cut-decompose command line option
    CUTSET, 53
    OUTPUT, 53
lhotse-cut-describe command line option
    CUTSET, 54
lhotse-cut-export-to-webdataset command line option
    --audio, 54
    --audio-format <audio_format>, 54
    --custom, 54
    --fault-tolerant, 54
    --features, 54
    --no-audio, 54

--no-custom, 54
--no-features, 54
--shard-size <shard_size>, 54
--stop-on-fail, 54
-f, 54
-s, 54
CUTSET, 54
WSPECIFIER, 54
lhotse-cut-mix-by-recording-id command line option
CUT_MANIFESTS, 55
OUTPUT_CUT_MANIFEST, 55
lhotse-cut-mix-sequential command line option
CUT_MANIFESTS, 55
OUTPUT_CUT_MANIFEST, 55
lhotse-cut-pad command line option
--duration <duration>, 55
-d, 55
CUT_MANIFEST, 56
OUTPUT_CUT_MANIFEST, 56
lhotse-cut-random-mixed command line option
--offset-range <offset_range>, 56
--snr-range <snr_range>, 56
-o, 56
-s, 56
FEATURE_MANIFEST, 56
OUTPUT_CUT_MANIFEST, 56
SUPERVISION_MANIFEST, 56
lhotse-cut-simple command line option
--feature-manifest <feature_manifest>, 57
--recording-manifest <recording_manifest>, 57
--supervision-manifest <supervision_manifest>, 57
-f, 57
-r, 57
-s, 57
OUTPUT_CUT_MANIFEST, 57
lhotse-cut-trim-to-supervisions command line option
--context-direction <context_direction>, 57
--discard-overlapping, 57
--keep-overlapping, 57
--min-duration <min_duration>, 57
-c, 57
-d, 57
CUTS, 58
OUTPUT_CUTS, 58
lhotse-cut-truncate command line option
--discard-overflowing-supervisions, 58
--keep-overflowing-supervisions, 58
--max-duration <max_duration>, 58
--offset-type <offset_type>, 58
--preserve-id, 58
-d, 58
-o, 58
CUT_MANIFEST, 58
OUTPUT_CUT_MANIFEST, 58
lhotse-cut-windowed command line option
--cut-duration <cut_duration>, 59
--cut-shift <cut_shift>, 59
--discard-shorter-windows, 59
--keep-shorter-windows, 59
-d, 59
-s, 59
FEATURE_MANIFEST, 59
OUTPUT_CUT_MANIFEST, 59
lhotse-download-adept command line option
TARGET_DIR, 59
lhotse-download-aidatatang-200zh command line option
TARGET_DIR, 60
lhotse-download-aishell command line option
TARGET_DIR, 60
lhotse-download-aishell4 command line option
TARGET_DIR, 60
lhotse-download-ali-meeting command line option
--force-download, 61
TARGET_DIR, 61
lhotse-download-ami command line option
--annotations <annotations>, 61
--force-download <force_download>, 61
--mic <mic>, 61
--url <url>, 61
TARGET_DIR, 61
lhotse-download-cmu-arctic command line option
TARGET_DIR, 62
lhotse-download-cmu-indic command line option
TARGET_DIR, 62
lhotse-download-earnings21 command line option
TARGET_DIR, 62
lhotse-download-gigaspeech command line option
--host <host>, 63
--subset <subset>, 63
PASSWORD, 63
TARGET_DIR, 63
lhotse-download-heroico command line option
TARGET_DIR, 63
lhotse-download-hifitts command line option
TARGET_DIR, 63

lhotse-download-icsi command line option
- --force-download <force_download>, 64
- --mic <mic>, 64
- --transcripts-dir <transcripts_dir>, 64
- --url <url>, 64
- AUDIO_DIR, 64

lhotse-download-libricss command line option
- --force-download, 64
- TARGET_DIR, 64

lhotse-download-librimix command line option
- TARGET_DIR, 65

lhotse-download-librispeech command line option
- --full, 65
- --mini, 65
- TARGET_DIR, 65

lhotse-download-libritts command line option
- TARGET_DIR, 65

lhotse-download-ljspeech command line option
- TARGET_DIR, 66

lhotse-download-mtedx command line option
- --lang <lang>, 66
- -l, 66
- TARGET_DIR, 66

lhotse-download-musan command line option
- TARGET_DIR, 66

lhotse-download-rir-noise command line option
- TARGET_DIR, 66

lhotse-download-spgispeech command line option
- TARGET_DIR, 67

lhotse-download-tedlium command line option
- TARGET_DIR, 67

lhotse-download-timit command line option
- TARGET_DIR, 67

lhotse-download-vctk command line option
- TARGET_DIR, 68

lhotse-download-voxceleb1 command line option
- --force-download, 68
- TARGET_DIR, 68

lhotse-download-voxceleb2 command line option
- --force-download, 68
- TARGET_DIR, 68

lhotse-download-yesno command line option
- TARGET_DIR, 69

lhotse-feat-extract command line option
- --feature-manifest <feature_manifest>, 69
- --lilcom-tick-power <lilcom_tick_power>, 69
- --num-jobs <num_jobs>, 69
- --root-dir <root_dir>, 69
- --storage-type <storage_type>, 69
- -f, 69
- -j, 69
- -r, 69
- -t, 69
- OUTPUT_DIR, 69
- RECORDING_MANIFEST, 69

lhotse-feat-extract-cuts command line option
- --feature-manifest <feature_manifest>, 70
- --num-jobs <num_jobs>, 70
- --storage-type <storage_type>, 70
- -f, 70
- -j, 70
- CUTSET, 70
- OUTPUT_CUTSET, 70
- STORAGE_PATH, 70

lhotse-feat-extract-cuts-batch command line option
- --batch-duration <batch_duration>, 71
- --feature-manifest <feature_manifest>, 71
- --num-jobs <num_jobs>, 71
- --storage-type <storage_type>, 71
- -b, 71
- -f, 71
- -j, 71
- CUTSET, 71
- OUTPUT_CUTSET, 71
- STORAGE_PATH, 71

lhotse-feat-upload command line option
- --num-jobs <num_jobs>, 71
- -j, 71
- FEATURE_MANIFEST, 72
- OUTPUT_MANIFEST, 72
- URL, 72

lhotse-feat-write-default-config command line option
- --feature-type <feature_type>, 72
- -f, 72
- OUTPUT_CONFIG, 72

lhotse-filter command line option
- MANIFEST, 73
- OUTPUT_MANIFEST, 73
- PREDICATE, 73

lhotse-fix command line option
- OUTPUT_DIR, 73
- RECORDINGS, 73
- SUPERVISIONS, 73

lhotse-install-sph2pipe command line option
- --install-dir <install_dir>, 73

--url <url>, 73
lhotse-kaldi-export command line option
--map-underscores-to <map_underscores_to>, 74
--prefix-spk-id, 74
-p, 74
-u, 74
OUTPUT_DIR, 74
RECORDINGS, 74
SUPERVISIONS, 74
lhotse-kaldi-import command line option
--frame-shift <frame_shift>, 74
--map-string-to-underscores <map_string_to_underscores>, 74
--num-jobs <num_jobs>, 74
-f, 74
-j, 74
-u, 74
DATA_DIR, 75
MANIFEST_DIR, 75
SAMPLING_RATE, 75
lhotse-prepare-adept command line option
CORPUS_DIR, 75
OUTPUT_DIR, 75
lhotse-prepare-aidatatang-200zh command line option
CORPUS_DIR, 75
OUTPUT_DIR, 75
lhotse-prepare-aishell command line option
CORPUS_DIR, 76
OUTPUT_DIR, 76
lhotse-prepare-aishell4 command line option
CORPUS_DIR, 76
OUTPUT_DIR, 76
lhotse-prepare-ali-meeting command line option
--mic <mic>, 76
CORPUS_DIR, 76
OUTPUT_DIR, 76
lhotse-prepare-ami command line option
--annotations <annotations>, 77
--mic <mic>, 77
--normalize-text <normalize_text>, 77
--partition <partition>, 77
CORPUS_DIR, 77
OUTPUT_DIR, 77
lhotse-prepare-aspire command line option
--mic <mic>, 77
CORPUS_DIR, 78
OUTPUT_DIR, 78
lhotse-prepare-babel command line option
CORPUS_DIR, 78
OUTPUT_DIR, 78
lhotse-prepare-broadcast-news command line option
AUDIO_DIR, 78
OUTPUT_DIR, 78
TRANSCRIPT_DIR, 78
lhotse-prepare-bvcc command line option
--num_jobs <num_jobs>, 79
-nj, 79
CORPUS_DIR, 79
OUTPUT_DIR, 79
lhotse-prepare-callhome-egyptian command line option
--absolute-paths <absolute_paths>, 80
AUDIO_DIR, 80
OUTPUT_DIR, 80
TRANSCRIPT_DIR, 80
lhotse-prepare-callhome-english command line option
--absolute-paths <absolute_paths>, 81
--rttm-dir <rttm_dir>, 81
--transcript-dir <transcript_dir>, 81
AUDIO_DIR, 81
OUTPUT_DIR, 81
lhotse-prepare-cmu-arctic command line option
CORPUS_DIR, 81
OUTPUT_DIR, 81
lhotse-prepare-cmu-indic command line option
CORPUS_DIR, 81
OUTPUT_DIR, 81
lhotse-prepare-cmu-kids command line option
--absolute-paths <absolute_paths>, 82
CORPUS_DIR, 82
OUTPUT_DIR, 82
lhotse-prepare-commonvoice command line option
--language <language>, 82
--num-jobs <num_jobs>, 82
--split <split>, 82
-j, 82
-l, 82
-s, 82
CORPUS_DIR, 82
OUTPUT_DIR, 82
lhotse-prepare-cslu-kids command line option
--absolute-paths <absolute_paths>, 83
--normalize-text <normalize_text>, 83
CORPUS_DIR, 83
OUTPUT_DIR, 83
lhotse-prepare-dihard3 command line option
--dev <dev>, 83
--eval <eval>, 83

--no-uem, 83
--num-jobs <num_jobs>, 83
--uem, 83
-j, 83
OUTPUT_DIR, 83

lhotse-prepare-earnings21 command line option
--no-normalize-text, 84
--normalize-text, 84
CORPUS_DIR, 84
OUTPUT_DIR, 84

lhotse-prepare-earnings22 command line option
--no-normalize-text, 84
--normalize-text, 84
CORPUS_DIR, 84
OUTPUT_DIR, 84

lhotse-prepare-eval2000 command line option
--absolute-paths <absolute_paths>, 85
CORPUS_DIR, 85
OUTPUT_DIR, 85

lhotse-prepare-fisher-english command line option
--absolute-paths <absolute_paths>, 85
--audio-dirs <audio_dirs>, 85
--num-jobs <num_jobs>, 85
--transcript-dirs <transcript_dirs>, 85
-ad, 85
-j, 85
-td, 85
CORPUS_DIR, 85
OUTPUT_DIR, 85

lhotse-prepare-fisher-spanish command line option
--absolute-paths <absolute_paths>, 86
AUDIO_DIR, 86
OUTPUT_DIR, 86
TRANSCRIPT_DIR, 86

lhotse-prepare-gale-arabic command line option
--absolute-paths <absolute_paths>, 86
--audio <audio>, 86
--transcript <transcript>, 86
-s, 86
-t, 86
OUTPUT_DIR, 87

lhotse-prepare-gale-mandarin command line option
--absolute-paths <absolute_paths>, 87
--audio <audio>, 87
--segment-words <segment_words>, 87
--transcript <transcript>, 87
-s, 87
-t, 87
OUTPUT_DIR, 87

lhotse-prepare-gigaspeech command line option
--num-jobs <num_jobs>, 87
--subset <subset>, 87
-j, 87
CORPUS_DIR, 88
OUTPUT_DIR, 88

lhotse-prepare-heroico command line option
OUTPUT_DIR, 88
SPEECH_DIR, 88
TRANSCRIPT_DIR, 88

lhotse-prepare-hifitts command line option
--num-jobs <num_jobs>, 88
-j, 88
CORPUS_DIR, 88
OUTPUT_DIR, 88

lhotse-prepare-icsi command line option
--mic <mic>, 89
--normalize-text, 89
--transcripts-dir <transcripts_dir>, 89
AUDIO_DIR, 89
OUTPUT_DIR, 89

lhotse-prepare-l2-arctic command line option
CORPUS_DIR, 89
OUTPUT_DIR, 89

lhotse-prepare-libricss command line option
--type <type>, 90
CORPUS_DIR, 90
OUTPUT_DIR, 90

lhotse-prepare-librimix command line option
--min-segment-seconds <min_segment_seconds>, 90
--no-precomputed-mixtures, 90
--sampling-rate <sampling_rate>, 90
--with-precomputed-mixtures, 90
LIBRIMIX_CSV, 90
OUTPUT_DIR, 90

lhotse-prepare-librispeech command line option
--dataset-parts <dataset_parts>, 91
--num-jobs <num_jobs>, 91
-j, 91
-p, 91
CORPUS_DIR, 91
OUTPUT_DIR, 91

lhotse-prepare-libritts command line option
--link-previous-utterance, 91
--no-previous-utterance, 91
--num-jobs <num_jobs>, 91
-j, 91
CORPUS_DIR, 91
OUTPUT_DIR, 91

lhotse-prepare-ljspeech command line option
- CORPUS_DIR, 92
- OUTPUT_DIR, 92

lhotse-prepare-mgb2 command line option
- --buck-walter, 92
- --mer-thresh <mer_thresh>, 92
- --no-buck-walter, 92
- --no-text-cleaning, 92
- --num-jobs <num_jobs>, 92
- --text-cleaning, 92
- -j, 92
- CORPUS_DIR, 92
- OUTPUT_DIR, 92

lhotse-prepare-mls command line option
- --flac, 93
- --num-jobs <num_jobs>, 93
- --opus, 93
- -j, 93
- CORPUS_DIR, 93
- OUTPUT_DIR, 93

lhotse-prepare-mtedx command line option
- --lang <lang>, 93
- --num-jobs <num_jobs>, 93
- -j, 93
- -l, 93
- CORPUS_DIR, 93
- OUTPUT_DIR, 93

lhotse-prepare-musan command line option
- --no-vocals, 94
- --use-vocals, 94
- CORPUS_DIR, 94
- OUTPUT_DIR, 94

lhotse-prepare-nsc command line option
- --dataset-part <dataset_part>, 94
- -p, 94
- CORPUS_DIR, 94
- OUTPUT_DIR, 94

lhotse-prepare-peoples-speech command line option
- CORPUS_DIR, 94
- OUTPUT_DIR, 94

lhotse-prepare-rir-noise command line option
- --parts <parts>, 95
- -p, 95
- CORPUS_DIR, 95
- OUTPUT_DIR, 95

lhotse-prepare-spgispeech command line option
- --no-normalize-text, 95
- --normalize-text, 95
- --num-jobs <num_jobs>, 95
- -j, 95
- CORPUS_DIR, 95
- OUTPUT_DIR, 95

lhotse-prepare-switchboard command line option
- --absolute-paths <absolute_paths>, 96
- --omit-silence, 96
- --retain-silence, 96
- --sentiment-dir <sentiment_dir>, 96
- --transcript-dir <transcript_dir>, 96
- AUDIO_DIR, 96
- OUTPUT_DIR, 96

lhotse-prepare-tedlium command line option
- OUTPUT_DIR, 96
- TEDLIUM_DIR, 96

lhotse-prepare-timit command line option
- --num-jobs <num_jobs>, 97
- --num-phones <num_phones>, 97
- -j, 97
- -p, 97
- CORPUS_DIR, 97
- OUTPUT_DIR, 97

lhotse-prepare-vctk command line option
- CORPUS_DIR, 97
- OUTPUT_DIR, 97

lhotse-prepare-voxceleb command line option
- --num-jobs <num_jobs>, 98
- --voxceleb1 <voxceleb1>, 98
- --voxceleb2 <voxceleb2>, 98
- -j, 98
- -v1, 98
- -v2, 98
- OUTPUT_DIR, 98

lhotse-prepare-wenet-speech command line option
- --dataset-parts <dataset_parts>, 98
- --num-jobs <num_jobs>, 98
- -j, 98
- -p, 98
- CORPUS_DIR, 98
- OUTPUT_DIR, 98

lhotse-prepare-yesno command line option
- CORPUS_DIR, 99
- OUTPUT_DIR, 99

lhotse-split command line option
- --no-pad, 99
- --pad, 99
- --shuffle, 99
- -s, 99
- MANIFEST, 99
- NUM_SPLITS, 99
- OUTPUT_DIR, 99

lhotse-split-lazy command line option
- CHUNK_SIZE, 100
- MANIFEST, 100
- OUTPUT_DIR, 100

lhotse-subset command line option
--cutids <cutids>, 100
--first <first>, 100
--last <last>, 100
MANIFEST, 100
OUTPUT_MANIFEST, 100
lhotse-validate command line option
--dont-read-data, 100
--read-data, 100
MANIFEST, 101
lhotse-validate-pair command line option
--dont-read-data, 101
--read-data, 101
RECORDINGS, 101
SUPERVISIONS, 101
LIBRIMIX_CSV
lhotse-prepare-librimix command line option, 90
LibrosaFbank (*class in lhotse.features.librosa_fbank*), 237
LibrosaFbankConfig (*class in lhotse.features.librosa_fbank*), 236
LibsndfileCompatibleAudioInfo (*class in lhotse.audio*), 116
LilcomChunkyReader (*class in lhotse.features.io*), 248
LilcomChunkyWriter (*class in lhotse.features.io*), 248
LilcomFilesReader (*class in lhotse.features.io*), 242
LilcomFilesWriter (*class in lhotse.features.io*), 242
LilcomHdf5Reader (*class in lhotse.features.io*), 245
LilcomHdf5Writer (*class in lhotse.features.io*), 245
LilcomURLReader (*class in lhotse.features.io*), 249
LilcomURLWriter (*class in lhotse.features.io*), 249
lin2mel() (*in module lhotse.features.kaldi.layers*), 225
load() (*lhotse.features.base.Features method*), 136
load() (*lhotse.features.base.FeatureSet method*), 139
load_audio (*lhotse.cut.Cut attribute*), 257
load_audio() (*lhotse.audio.AudioSource method*), 104
load_audio() (*lhotse.audio.Recording method*), 107
load_audio() (*lhotse.audio.RecordingSet method*), 111
load_audio() (*lhotse.cut.MixedCut method*), 286
load_audio() (*lhotse.cut.MonoCut method*), 264
load_audio() (*lhotse.cut.PaddingCut method*), 274
load_custom() (*lhotse.cut.MixedCut method*), 283
load_custom() (*lhotse.cut.MonoCut method*), 263
load_features (*lhotse.cut.Cut attribute*), 257
load_features() (*lhotse.cut.MixedCut method*), 286
load_features() (*lhotse.cut.MonoCut method*), 263
load_features() (*lhotse.cut.PaddingCut method*), 274
load_kaldi_data_dir() (*in module lhotse.kaldi*), 316
load_kaldi_text_mapping() (*in module lhotse.kaldi*), 317
load_state_dict() (*lhotse.dataset.signal_transforms.SpecAugment method*), 42
load_state_dict() (*lhotse.features.kaldi.layers.Wav2FFT method*), 163
load_state_dict() (*lhotse.features.kaldi.layers.Wav2LogFilterBank method*), 203
load_state_dict() (*lhotse.features.kaldi.layers.Wav2LogSpec method*), 190
load_state_dict() (*lhotse.features.kaldi.layers.Wav2MFCC method*), 217
load_state_dict() (*lhotse.features.kaldi.layers.Wav2Spec method*), 176
load_state_dict() (*lhotse.features.kaldi.layers.Wav2Win method*), 149
logmelfilterbank() (*in module lhotse.features.librosa_fbank*), 236
lookup_cache_or_open() (*in module lhotse.features.io*), 244
lookup_cache_or_open_regular_file() (*in module lhotse.features.io*), 244
lookup_chunk_size() (*in module lhotse.features.io*), 244
low_freq (*lhotse.features.fbank.TorchaudioFbankConfig attribute*), 226
low_freq (*lhotse.features.mfcc.TorchaudioMfccConfig attribute*), 230

M

make_dct_matrix() (*lhotse.features.kaldi.layers.Wav2MFCC static method*), 212
make_lifter() (*lhotse.features.kaldi.layers.Wav2MFCC static method*), 212
make_wavscp_channel_string_map() (*in module lhotse.kaldi*), 317
make_windowed_cuts_from_features() (*in module lhotse.cut*), 313
MANIFEST
lhotse-filter command line option, 73
lhotse-split command line option, 99
lhotse-split-lazy command line option, 100
lhotse-subset command line option, 100
lhotse-validate command line option, 101
MANIFEST_DIR
lhotse-kaldi-import command line option, 75
MANIFESTS
lhotse-combine command line option, 51
map() (*lhotse.audio.RecordingSet method*), 114
map() (*lhotse.cut.CutSet method*), 311
map() (*lhotse.features.base.FeatureSet method*), 140
map() (*lhotse.supervision.SupervisionSegment method*), 122
map() (*lhotse.supervision.SupervisionSet method*), 128
map_supervisions (*lhotse.cut.Cut attribute*), 258
map_supervisions() (*lhotse.cut.CutSet method*), 310

map_supervisions() (*lhotse.cut.MixedCut method*), 287
map_supervisions() (*lhotse.cut.MonoCut method*), 268
map_supervisions() (*lhotse.cut.PaddingCut method*), 276
maybe_pad() (*in module lhotse.dataset.collation*), 46
mel2lin() (*in module lhotse.features.kaldi.layers*), 225
MemoryLilcomReader (*class in lhotse.features.io*), 252
MemoryLilcomWriter (*class in lhotse.features.io*), 252
MemoryRawReader (*class in lhotse.features.io*), 253
MemoryRawWriter (*class in lhotse.features.io*), 253
merge_supervisions (*lhotse.cut.Cut attribute*), 258
merge_supervisions() (*in module lhotse.cut*), 315
merge_supervisions() (*lhotse.cut.CutSet method*), 299
merge_supervisions() (*lhotse.cut.MixedCut method*), 287
merge_supervisions() (*lhotse.cut.MonoCut method*), 268
merge_supervisions() (*lhotse.cut.PaddingCut method*), 277
Mfcc (*class in lhotse.features.kaldi.extractors*), 143
min_duration (*lhotse.features.fbank.TorchaudioFbankConfig attribute*), 226
min_duration (*lhotse.features.mfcc.TorchaudioMfccConfig attribute*), 229
min_duration (*lhotse.features.spectrogram.SpectrogramConfig attribute*), 233
mix() (*in module lhotse.cut*), 313
mix() (*lhotse.cut.Cut method*), 259
mix() (*lhotse.cut.CutSet method*), 304
mix() (*lhotse.cut.MixedCut method*), 290
mix() (*lhotse.cut.MonoCut method*), 270
mix() (*lhotse.cut.PaddingCut method*), 278
mix() (*lhotse.features.base.FeatureExtractor static method*), 131
mix() (*lhotse.features.base.TorchaudioFeatureExtractor static method*), 135
mix() (*lhotse.features.fbank.TorchaudioFbank static method*), 227
mix() (*lhotse.features.kaldi.extractors.Fbank static method*), 143
mix() (*lhotse.features.librosa_fbank.LibrosaFbank static method*), 237
mix() (*lhotse.features.mfcc.TorchaudioMfcc static method*), 232
mix() (*lhotse.features.spectrogram.Spectrogram static method*), 233
mix_cuts() (*in module lhotse.cut*), 314
mix_same_recording_channels() (*lhotse.cut.CutSet method*), 301
mixed_audio (*lhotse.audio.AudioMixer property*), 116
mixed_cuts (*lhotse.cut.CutSet property*), 296
mixed_feats (*lhotse.features.mixer.FeatureMixer property*), 254
MixedCut (*class in lhotse.cut*), 282
MixTrack (*class in lhotse.cut*), 281
modify_ids() (*lhotse.cut.CutSet method*), 309
module
  lhotse.audio, 103
  lhotse.augmentation, 255
  lhotse.cut, 255
  lhotse.dataset.collation, 44
  lhotse.dataset.cut_transforms, 39
  lhotse.dataset.diarization, 31
  lhotse.dataset.input_strategies, 35
  lhotse.dataset.sampling, 35
  lhotse.dataset.signal_transforms, 41
  lhotse.dataset.speech_recognition, 33
  lhotse.dataset.unsupervised, 32
  lhotse.dataset.vad, 35
  lhotse.features.base, 130
  lhotse.features.fbank, 226
  lhotse.features.io, 240
  lhotse.features.librosa_fbank, 236
  lhotse.features.mfcc, 229
  lhotse.features.mixer, 254
  lhotse.features.spectrogram, 233
  lhotse.kaldi, 316
  lhotse.manipulation, 317
  lhotse.recipes, 316
  lhotse.supervision, 119
modules() (*lhotse.features.kaldi.layers.Wav2FFT method*), 163
modules() (*lhotse.features.kaldi.layers.Wav2LogFilterBank method*), 203
modules() (*lhotse.features.kaldi.layers.Wav2LogSpec method*), 190
modules() (*lhotse.features.kaldi.layers.Wav2MFCC method*), 217
modules() (*lhotse.features.kaldi.layers.Wav2Spec method*), 176
modules() (*lhotse.features.kaldi.layers.Wav2Win method*), 150
MonoCut (*class in lhotse.cut*), 262
move_to_memory() (*lhotse.audio.Recording method*), 106
move_to_memory() (*lhotse.cut.MonoCut method*), 264
move_to_memory() (*lhotse.features.base.Features method*), 136
mux() (*lhotse.audio.RecordingSet class method*), 114
mux() (*lhotse.cut.CutSet class method*), 311
mux() (*lhotse.features.base.FeatureSet class method*), 140
mux() (*lhotse.supervision.SupervisionSet class method*), 128

N

name (*lhotse.features.base.FeatureExtractor attribute*), 130
name (*lhotse.features.base.TorchaudioFeatureExtractor attribute*), 136
name (*lhotse.features.fbank.TorchaudioFbank attribute*), 226
name (*lhotse.features.io.ChunkedLilcomHdf5Reader attribute*), 246
name (*lhotse.features.io.ChunkedLilcomHdf5Writer attribute*), 247
name (*lhotse.features.io.FeaturesReader property*), 241
name (*lhotse.features.io.FeaturesWriter property*), 240
name (*lhotse.features.io.KaldiReader attribute*), 250
name (*lhotse.features.io.KaldiWriter attribute*), 251
name (*lhotse.features.io.LilcomChunkyReader attribute*), 248
name (*lhotse.features.io.LilcomChunkyWriter attribute*), 248
name (*lhotse.features.io.LilcomFilesReader attribute*), 242
name (*lhotse.features.io.LilcomFilesWriter attribute*), 242
name (*lhotse.features.io.LilcomHdf5Reader attribute*), 245
name (*lhotse.features.io.LilcomHdf5Writer attribute*), 245
name (*lhotse.features.io.LilcomURLReader attribute*), 249
name (*lhotse.features.io.LilcomURLWriter attribute*), 250
name (*lhotse.features.io.MemoryLilcomReader attribute*), 252
name (*lhotse.features.io.MemoryLilcomWriter attribute*), 252
name (*lhotse.features.io.MemoryRawReader attribute*), 253
name (*lhotse.features.io.MemoryRawWriter attribute*), 253
name (*lhotse.features.io.NumpyFilesReader attribute*), 243
name (*lhotse.features.io.NumpyFilesWriter attribute*), 243
name (*lhotse.features.io.NumpyHdf5Reader attribute*), 244
name (*lhotse.features.io.NumpyHdf5Writer attribute*), 244
name (*lhotse.features.kaldi.extractors.Fbank attribute*), 143
name (*lhotse.features.kaldi.extractors.Mfcc attribute*), 143
name (*lhotse.features.librosa_fbank.LibrosaFbank attribute*), 237
name (*lhotse.features.mfcc.TorchaudioMfcc attribute*), 230
name (*lhotse.features.spectrogram.Spectrogram attribute*), 233
named_buffers() (*lhotse.features.kaldi.layers.Wav2FFT method*), 164
named_buffers() (*lhotse.features.kaldi.layers.Wav2LogFilterBank method*), 204
named_buffers() (*lhotse.features.kaldi.layers.Wav2LogSpec method*), 190
named_buffers() (*lhotse.features.kaldi.layers.Wav2MFCC method*), 218
named_buffers() (*lhotse.features.kaldi.layers.Wav2Spec method*), 177
named_buffers() (*lhotse.features.kaldi.layers.Wav2Win method*), 150
named_children() (*lhotse.features.kaldi.layers.Wav2FFT method*), 164
named_children() (*lhotse.features.kaldi.layers.Wav2LogFilterBank method*), 204
named_children() (*lhotse.features.kaldi.layers.Wav2LogSpec method*), 191
named_children() (*lhotse.features.kaldi.layers.Wav2MFCC method*), 218
named_children() (*lhotse.features.kaldi.layers.Wav2Spec method*), 177
named_children() (*lhotse.features.kaldi.layers.Wav2Win method*), 151
named_modules() (*lhotse.features.kaldi.layers.Wav2FFT method*), 164
named_modules() (*lhotse.features.kaldi.layers.Wav2LogFilterBank method*), 204
named_modules() (*lhotse.features.kaldi.layers.Wav2LogSpec method*), 191
named_modules() (*lhotse.features.kaldi.layers.Wav2MFCC method*), 218
named_modules() (*lhotse.features.kaldi.layers.Wav2Spec method*), 177
named_modules() (*lhotse.features.kaldi.layers.Wav2Win method*), 151
named_parameters() (*lhotse.features.kaldi.layers.Wav2FFT method*), 165
named_parameters() (*lhotse.features.kaldi.layers.Wav2LogFilterBank method*), 205
named_parameters() (*lhotse.features.kaldi.layers.Wav2LogSpec method*), 191
named_parameters() (*lhotse.features.kaldi.layers.Wav2MFCC method*), 219
named_parameters() (*lhotse.features.kaldi.layers.Wav2Spec method*), 178
named_parameters() (*lhotse.features.kaldi.layers.Wav2Win method*), 151
next_power_of_2() (*in module lhotse.features.kaldi.layers*), 225
null_result_on_audio_loading_error() (*in module lhotse.audio*), 118
num_ceps (*lhotse.features.mfcc.TorchaudioMfccConfig attribute*), 230
num_channels (*lhotse.audio.Recording property*), 107
num_channels() (*lhotse.audio.RecordingSet method*),

111
num_features (*lhotse.cut.Cut attribute*), 257
num_features (*lhotse.cut.MixedCut property*), 283
num_features (*lhotse.cut.MonoCut property*), 263
num_features (*lhotse.cut.PaddingCut attribute*), 273
num_features (*lhotse.features.base.Features attribute*), 136
num_features (*lhotse.features.mixer.FeatureMixer property*), 254
num_frames (*lhotse.cut.Cut attribute*), 257
num_frames (*lhotse.cut.MixedCut property*), 283
num_frames (*lhotse.cut.MonoCut property*), 263
num_frames (*lhotse.cut.PaddingCut attribute*), 273
num_frames (*lhotse.features.base.Features attribute*), 136
num_mel_bins (*lhotse.features.fbank.TorchaudioFbankConfig attribute*), 226
num_mel_bins (*lhotse.features.librosa_fbank.LibrosaFbankConfig attribute*), 236
num_mel_bins (*lhotse.features.mfcc.TorchaudioMfccConfig attribute*), 230
num_samples (*lhotse.audio.Recording attribute*), 105
num_samples (*lhotse.cut.Cut attribute*), 257
num_samples (*lhotse.cut.MixedCut property*), 283
num_samples (*lhotse.cut.MonoCut property*), 263
num_samples (*lhotse.cut.PaddingCut attribute*), 274
num_samples() (*lhotse.audio.RecordingSet method*), 112
num_samples_total (*lhotse.audio.AudioMixer property*), 116
NUM_SPLITS
lhotse-split command line option, 99
NumpyFilesReader (*class in lhotse.features.io*), 243
NumpyFilesWriter (*class in lhotse.features.io*), 243
NumpyHdf5Reader (*class in lhotse.features.io*), 244
NumpyHdf5Writer (*class in lhotse.features.io*), 244

O

offset (*lhotse.cut.MixTrack attribute*), 282
online_inference() (*lhotse.features.kaldi.layers.Wav2FFT method*), 158
online_inference() (*lhotse.features.kaldi.layers.Wav2LogFilterBank method*), 205
online_inference() (*lhotse.features.kaldi.layers.Wav2LogSpec method*), 192
online_inference() (*lhotse.features.kaldi.layers.Wav2MFCC method*), 219
online_inference() (*lhotse.features.kaldi.layers.Wav2Spec method*), 178
online_inference() (*lhotse.features.kaldi.layers.Wav2Win method*), 145
OnTheFlyFeatures (*class in lhotse.dataset.input_strategies*), 37
open_writer() (*lhotse.audio.RecordingSet class method*), 114
open_writer() (*lhotse.cut.CutSet class method*), 311
open_writer() (*lhotse.features.base.FeatureSet class method*), 141
open_writer() (*lhotse.supervision.SupervisionSet class method*), 128
opus_info() (*in module lhotse.audio*), 117
OUTPUT
lhotse-cut-decompose command line option, 53
OUTPUT_CONFIG
lhotse-feat-write-default-config command line option, 72
OUTPUT_CUT_MANIFEST
lhotse-cut-append command line option, 53
lhotse-cut-mix-by-recording-id command line option, 55
lhotse-cut-mix-sequential command line option, 55
lhotse-cut-pad command line option, 56
lhotse-cut-random-mixed command line option, 56
lhotse-cut-simple command line option, 57
lhotse-cut-truncate command line option, 58
lhotse-cut-windowed command line option, 59
OUTPUT_CUTS
lhotse-cut-trim-to-supervisions command line option, 58
OUTPUT_CUTSET
lhotse-feat-extract-cuts command line option, 70
lhotse-feat-extract-cuts-batch command line option, 71
OUTPUT_DIR
lhotse-feat-extract command line option, 69
lhotse-fix command line option, 73
lhotse-kaldi-export command line option, 74
lhotse-prepare-adept command line option, 75
lhotse-prepare-aidatatang-200zh command line option, 75
lhotse-prepare-aishell command line option, 76
lhotse-prepare-aishell4 command line option, 76
lhotse-prepare-ali-meeting command line option, 76
lhotse-prepare-ami command line option, 77

lhotse-prepare-aspire command line option, 78
lhotse-prepare-babel command line option, 78
lhotse-prepare-broadcast-news command line option, 78
lhotse-prepare-bvcc command line option, 79
lhotse-prepare-callhome-egyptian command line option, 80
lhotse-prepare-callhome-english command line option, 81
lhotse-prepare-cmu-arctic command line option, 81
lhotse-prepare-cmu-indic command line option, 81
lhotse-prepare-cmu-kids command line option, 82
lhotse-prepare-commonvoice command line option, 82
lhotse-prepare-cslu-kids command line option, 83
lhotse-prepare-dihard3 command line option, 83
lhotse-prepare-earnings21 command line option, 84
lhotse-prepare-earnings22 command line option, 84
lhotse-prepare-eval2000 command line option, 85
lhotse-prepare-fisher-english command line option, 85
lhotse-prepare-fisher-spanish command line option, 86
lhotse-prepare-gale-arabic command line option, 87
lhotse-prepare-gale-mandarin command line option, 87
lhotse-prepare-gigaspeech command line option, 88
lhotse-prepare-heroico command line option, 88
lhotse-prepare-hifitts command line option, 88
lhotse-prepare-icsi command line option, 89
lhotse-prepare-l2-arctic command line option, 89
lhotse-prepare-libricss command line option, 90
lhotse-prepare-librimix command line option, 90
lhotse-prepare-librispeech command line option, 91
lhotse-prepare-libritts command line option, 91
lhotse-prepare-ljspeech command line option, 92
lhotse-prepare-mgb2 command line option, 92
lhotse-prepare-mls command line option, 93
lhotse-prepare-mtedx command line option, 93
lhotse-prepare-musan command line option, 94
lhotse-prepare-nsc command line option, 94
lhotse-prepare-peoples-speech command line option, 94
lhotse-prepare-rir-noise command line option, 95
lhotse-prepare-spgispeech command line option, 95
lhotse-prepare-switchboard command line option, 96
lhotse-prepare-tedlium command line option, 96
lhotse-prepare-timit command line option, 97
lhotse-prepare-vctk command line option, 97
lhotse-prepare-voxceleb command line option, 98
lhotse-prepare-wenet-speech command line option, 98
lhotse-prepare-yesno command line option, 99
lhotse-split command line option, 99
lhotse-split-lazy command line option, 100

OUTPUT_MANIFEST
lhotse-combine command line option, 51
lhotse-copy command line option, 52
lhotse-copy-feats command line option, 52
lhotse-feat-upload command line option, 72
lhotse-filter command line option, 73
lhotse-subset command line option, 100

P

pad (*lhotse.cut.Cut attribute*), 258
pad() (*in module lhotse.cut*), 313
pad() (*lhotse.cut.CutSet method*), 301
pad() (*lhotse.cut.MixedCut method*), 284
pad() (*lhotse.cut.MonoCut method*), 266
pad() (*lhotse.cut.PaddingCut method*), 274

pad_or_truncate_features() (*in module lhotse.features.librosa_fbank*), 236
PaddingCut (*class in lhotse.cut*), 273
pairwise() (*in module lhotse.features.io*), 254
parameters() (*lhotse.features.kaldi.layers.Wav2FFT method*), 165
parameters() (*lhotse.features.kaldi.layers.Wav2LogFilterBank method*), 205
parameters() (*lhotse.features.kaldi.layers.Wav2LogSpec method*), 192
parameters() (*lhotse.features.kaldi.layers.Wav2MFCC method*), 219
parameters() (*lhotse.features.kaldi.layers.Wav2Spec method*), 178
parameters() (*lhotse.features.kaldi.layers.Wav2Win method*), 152
parse_channel_from_ffmpeg_output() (*in module lhotse.audio*), 118
PASSWORD
lhotse-download-gigaspeech command line option, 63
perturb_speed (*lhotse.cut.Cut attribute*), 258
perturb_speed() (*lhotse.audio.Recording method*), 107
perturb_speed() (*lhotse.audio.RecordingSet method*), 112
perturb_speed() (*lhotse.cut.CutSet method*), 303
perturb_speed() (*lhotse.cut.MixedCut method*), 285
perturb_speed() (*lhotse.cut.MonoCut method*), 267
perturb_speed() (*lhotse.cut.PaddingCut method*), 275
perturb_speed() (*lhotse.supervision.AlignmentItem method*), 119
perturb_speed() (*lhotse.supervision.SupervisionSegment method*), 121
perturb_tempo (*lhotse.cut.Cut attribute*), 258
perturb_tempo() (*lhotse.audio.Recording method*), 107
perturb_tempo() (*lhotse.audio.RecordingSet method*), 112
perturb_tempo() (*lhotse.cut.CutSet method*), 303
perturb_tempo() (*lhotse.cut.MixedCut method*), 285
perturb_tempo() (*lhotse.cut.MonoCut method*), 267
perturb_tempo() (*lhotse.cut.PaddingCut method*), 275
perturb_tempo() (*lhotse.supervision.SupervisionSegment method*), 122
perturb_volume (*lhotse.cut.Cut attribute*), 258
perturb_volume() (*lhotse.audio.Recording method*), 108
perturb_volume() (*lhotse.audio.RecordingSet method*), 112
perturb_volume() (*lhotse.cut.CutSet method*), 304
perturb_volume() (*lhotse.cut.MixedCut method*), 285
perturb_volume() (*lhotse.cut.MonoCut method*), 267
perturb_volume() (*lhotse.cut.PaddingCut method*), 276
perturb_volume() (*lhotse.supervision.SupervisionSegment method*), 122
PerturbSpeed (*class in lhotse.dataset.cut_transforms*), 40
PerturbTempo (*class in lhotse.dataset.cut_transforms*), 40
PerturbVolume (*class in lhotse.dataset.cut_transforms*), 40
play_audio() (*lhotse.cut.Cut method*), 259
play_audio() (*lhotse.cut.MixedCut method*), 290
play_audio() (*lhotse.cut.MonoCut method*), 270
play_audio() (*lhotse.cut.PaddingCut method*), 279
plot_alignment() (*lhotse.cut.Cut method*), 259
plot_alignment() (*lhotse.cut.MixedCut method*), 290
plot_alignment() (*lhotse.cut.MonoCut method*), 270
plot_alignment() (*lhotse.cut.PaddingCut method*), 279
plot_audio() (*lhotse.cut.Cut method*), 259
plot_audio() (*lhotse.cut.MixedCut method*), 290
plot_audio() (*lhotse.cut.MonoCut method*), 270
plot_audio() (*lhotse.cut.PaddingCut method*), 279
plot_features() (*lhotse.cut.Cut method*), 259
plot_features() (*lhotse.cut.MixedCut method*), 290
plot_features() (*lhotse.cut.MonoCut method*), 270
plot_features() (*lhotse.cut.PaddingCut method*), 279
plot_tracks_audio() (*lhotse.cut.MixedCut method*), 286
plot_tracks_features() (*lhotse.cut.MixedCut method*), 286
PrecomputedFeatures (*class in lhotse.dataset.input_strategies*), 36
PREDICATE
lhotse-filter command line option, 73
preemph_coeff (*lhotse.features.kaldi.layers.Wav2FFT property*), 158
preemph_coeff (*lhotse.features.kaldi.layers.Wav2LogFilterBank property*), 206
preemph_coeff (*lhotse.features.kaldi.layers.Wav2LogSpec property*), 192
preemph_coeff (*lhotse.features.kaldi.layers.Wav2MFCC property*), 219
preemph_coeff (*lhotse.features.kaldi.layers.Wav2Spec property*), 179
preemphasis_coefficient (*lhotse.features.fbank.TorchaudioFbankConfig attribute*), 226
preemphasis_coefficient (*lhotse.features.mfcc.TorchaudioMfccConfig attribute*), 229
preemphasis_coefficient (*lhotse.features.spectrogram.SpectrogramConfig attribute*), 233
PreMixedSourceSeparationDataset (*class in*

lhotse.dataset.source_separation), 34
process_and_store_recordings() (lhotse.features.base.FeatureSetBuilder method), 142

R

RandomizedSmoothing (class in lhotse.dataset.signal_transforms), 43
raw_energy (lhotse.features.fbank.TorchaudioFbankConfig attribute), 226
raw_energy (lhotse.features.mfcc.TorchaudioMfccConfig attribute), 229
raw_energy (lhotse.features.spectrogram.SpectrogramConfig attribute), 233
read() (lhotse.features.io.ChunkedLilcomHdf5Reader method), 247
read() (lhotse.features.io.FeaturesReader method), 241
read() (lhotse.features.io.KaldiReader method), 250
read() (lhotse.features.io.LilcomChunkyReader method), 248
read() (lhotse.features.io.LilcomFilesReader method), 242
read() (lhotse.features.io.LilcomHdf5Reader method), 245
read() (lhotse.features.io.LilcomURLReader method), 249
read() (lhotse.features.io.MemoryLilcomReader method), 252
read() (lhotse.features.io.MemoryRawReader method), 253
read() (lhotse.features.io.NumpyFilesReader method), 243
read() (lhotse.features.io.NumpyHdf5Reader method), 244
read_audio() (in module lhotse.audio), 116
read_audio_from_cuts() (in module lhotse.dataset.collation), 46
read_features_from_cuts() (in module lhotse.dataset.collation), 47
read_opus() (in module lhotse.audio), 117
read_opus_ffmpeg() (in module lhotse.audio), 118
read_opus_torchaudio() (in module lhotse.audio), 117
read_sph() (in module lhotse.audio), 118
Recording (class in lhotse.audio), 104
recording (lhotse.cut.MonoCut attribute), 263
recording_id (lhotse.cut.MonoCut property), 263
recording_id (lhotse.cut.PaddingCut property), 274
recording_id (lhotse.features.base.Features attribute), 136
recording_id (lhotse.supervision.SupervisionSegment attribute), 121
RECORDING_MANIFEST
lhotse-feat-extract command line option, 69
RECORDINGS
lhotse-fix command line option, 73
lhotse-kaldi-export command line option, 74
lhotse-validate-pair command line option, 101
RecordingSet (class in lhotse.audio), 108
register_backward_hook() (lhotse.features.kaldi.layers.Wav2FFT method), 165
register_backward_hook() (lhotse.features.kaldi.layers.Wav2LogFilterBank method), 206
register_backward_hook() (lhotse.features.kaldi.layers.Wav2LogSpec method), 192
register_backward_hook() (lhotse.features.kaldi.layers.Wav2MFCC method), 219
register_backward_hook() (lhotse.features.kaldi.layers.Wav2Spec method), 179
register_backward_hook() (lhotse.features.kaldi.layers.Wav2Win method), 152
register_buffer() (lhotse.features.kaldi.layers.Wav2FFT method), 166
register_buffer() (lhotse.features.kaldi.layers.Wav2LogFilterBank method), 206
register_buffer() (lhotse.features.kaldi.layers.Wav2LogSpec method), 192
register_buffer() (lhotse.features.kaldi.layers.Wav2MFCC method), 220
register_buffer() (lhotse.features.kaldi.layers.Wav2Spec method), 179
register_buffer() (lhotse.features.kaldi.layers.Wav2Win method), 152
register_extractor() (in module lhotse.features.base), 133
register_forward_hook() (lhotse.features.kaldi.layers.Wav2FFT method), 166
register_forward_hook() (lhotse.features.kaldi.layers.Wav2LogFilterBank method), 206
register_forward_hook() (lhotse.features.kaldi.layers.Wav2LogSpec method), 193
register_forward_hook() (lhotse.features.kaldi.layers.Wav2MFCC method), 220
register_forward_hook()

(*lhotse.features.kaldi.layers.Wav2Spec method*), 179
register_forward_hook() (*lhotse.features.kaldi.layers.Wav2Win method*), 153
register_forward_pre_hook() (*lhotse.features.kaldi.layers.Wav2FFT method*), 166
register_forward_pre_hook() (*lhotse.features.kaldi.layers.Wav2LogFilterBank method*), 207
register_forward_pre_hook() (*lhotse.features.kaldi.layers.Wav2LogSpec method*), 193
register_forward_pre_hook() (*lhotse.features.kaldi.layers.Wav2MFCC method*), 220
register_forward_pre_hook() (*lhotse.features.kaldi.layers.Wav2Spec method*), 180
register_forward_pre_hook() (*lhotse.features.kaldi.layers.Wav2Win method*), 153
register_full_backward_hook() (*lhotse.features.kaldi.layers.Wav2FFT method*), 167
register_full_backward_hook() (*lhotse.features.kaldi.layers.Wav2LogFilterBank method*), 207
register_full_backward_hook() (*lhotse.features.kaldi.layers.Wav2LogSpec method*), 193
register_full_backward_hook() (*lhotse.features.kaldi.layers.Wav2MFCC method*), 221
register_full_backward_hook() (*lhotse.features.kaldi.layers.Wav2Spec method*), 180
register_full_backward_hook() (*lhotse.features.kaldi.layers.Wav2Win method*), 153
register_module() (*lhotse.features.kaldi.layers.Wav2FFT method*), 167
register_module() (*lhotse.features.kaldi.layers.Wav2LogFilterBank method*), 207
register_module() (*lhotse.features.kaldi.layers.Wav2LogSpec method*), 194
register_module() (*lhotse.features.kaldi.layers.Wav2MFCC method*), 221
register_module() (*lhotse.features.kaldi.layers.Wav2Spec method*), 180
register_module() (*lhotse.features.kaldi.layers.Wav2Win method*), 154
register_parameter() (*lhotse.features.kaldi.layers.Wav2FFT method*), 167
register_parameter() (*lhotse.features.kaldi.layers.Wav2LogFilterBank method*), 207
register_parameter() (*lhotse.features.kaldi.layers.Wav2LogSpec method*), 194
register_parameter() (*lhotse.features.kaldi.layers.Wav2MFCC method*), 221
register_parameter() (*lhotse.features.kaldi.layers.Wav2Spec method*), 180
register_parameter() (*lhotse.features.kaldi.layers.Wav2Win method*), 154
register_reader() (*in module lhotse.features.io*), 241
register_writer() (*in module lhotse.features.io*), 241
remove_dc_offset (*lhotse.features.fbank.TorchaudioFbankConfig attribute*), 226
remove_dc_offset (*lhotse.features.kaldi.layers.Wav2FFT property*), 158
remove_dc_offset (*lhotse.features.kaldi.layers.Wav2LogFilterBank property*), 208
remove_dc_offset (*lhotse.features.kaldi.layers.Wav2LogSpec property*), 194
remove_dc_offset (*lhotse.features.kaldi.layers.Wav2MFCC property*), 222
remove_dc_offset (*lhotse.features.kaldi.layers.Wav2Spec property*), 181
remove_dc_offset (*lhotse.features.mfcc.TorchaudioMfccConfig attribute*), 229
remove_dc_offset (*lhotse.features.spectrogram.SpectrogramConfig attribute*), 233
repeat() (*lhotse.audio.RecordingSet method*), 115
repeat() (*lhotse.cut.CutSet method*), 312
repeat() (*lhotse.features.base.FeatureSet method*), 141
repeat() (*lhotse.supervision.SupervisionSet method*), 129
requires_grad_() (*lhotse.features.kaldi.layers.Wav2FFT method*), 168
requires_grad_() (*lhotse.features.kaldi.layers.Wav2LogFilterBank method*), 208
requires_grad_() (*lhotse.features.kaldi.layers.Wav2LogSpec method*), 194
requires_grad_() (*lhotse.features.kaldi.layers.Wav2MFCC method*), 222
requires_grad_() (*lhotse.features.kaldi.layers.Wav2Spec method*), 181
requires_grad_() (*lhotse.features.kaldi.layers.Wav2Win method*), 154
resample (*lhotse.cut.Cut attribute*), 258
resample() (*lhotse.audio.Recording method*), 108

resample() (*lhotse.audio.RecordingSet method*), 113
resample() (*lhotse.cut.CutSet method*), 303
resample() (*lhotse.cut.MixedCut method*), 284
resample() (*lhotse.cut.MonoCut method*), 266
resample() (*lhotse.cut.PaddingCut method*), 275
reverb_rir (*lhotse.cut.Cut attribute*), 258
reverb_rir() (*lhotse.audio.Recording method*), 108
reverb_rir() (*lhotse.audio.RecordingSet method*), 112
reverb_rir() (*lhotse.cut.CutSet method*), 304
reverb_rir() (*lhotse.cut.MixedCut method*), 285
reverb_rir() (*lhotse.cut.MonoCut method*), 267
reverb_rir() (*lhotse.cut.PaddingCut method*), 276
reverb_rir() (*lhotse.supervision.SupervisionSegment method*), 122
ReverbWithImpulseResponse (*class in lhotse.dataset.cut_transforms*), 40
round_to_power_of_two (*lhotse.features.fbank.TorchaudioFbankConfig attribute*), 226
round_to_power_of_two (*lhotse.features.mfcc.TorchaudioMfccConfig attribute*), 229
round_to_power_of_two (*lhotse.features.spectrogram.SpectrogramConfig attribute*), 233

S

sample() (*lhotse.cut.CutSet method*), 303
samplerate (*lhotse.audio.LibsndfileCompatibleAudioInfo property*), 116
SAMPLING_RATE
    lhotse-kaldi-import command line option, 75
sampling_rate (*lhotse.audio.Recording attribute*), 105
sampling_rate (*lhotse.cut.Cut attribute*), 257
sampling_rate (*lhotse.cut.MixedCut property*), 283
sampling_rate (*lhotse.cut.MonoCut property*), 263
sampling_rate (*lhotse.cut.PaddingCut attribute*), 273
sampling_rate (*lhotse.features.base.Features attribute*), 136
sampling_rate (*lhotse.features.kaldi.layers.Wav2FFT property*), 158
sampling_rate (*lhotse.features.kaldi.layers.Wav2LogFilterBank property*), 208
sampling_rate (*lhotse.features.kaldi.layers.Wav2LogSpec property*), 195
sampling_rate (*lhotse.features.kaldi.layers.Wav2MFCC property*), 222
sampling_rate (*lhotse.features.kaldi.layers.Wav2Spec property*), 181
sampling_rate (*lhotse.features.librosa_fbank.LibrosaFbankConfig attribute*), 236
sampling_rate() (*lhotse.audio.RecordingSet method*), 112
save_audio() (*lhotse.cut.Cut method*), 261
save_audio() (*lhotse.cut.MixedCut method*), 290
save_audio() (*lhotse.cut.MonoCut method*), 270
save_audio() (*lhotse.cut.PaddingCut method*), 279
save_audios() (*lhotse.cut.CutSet method*), 308
save_kaldi_text_mapping() (*in module lhotse.kaldi*), 317
set_audio_duration_mismatch_tolerance() (*in module lhotse.audio*), 103
set_extra_state() (*lhotse.features.kaldi.layers.Wav2FFT method*), 168
set_extra_state() (*lhotse.features.kaldi.layers.Wav2LogFilterBank method*), 208
set_extra_state() (*lhotse.features.kaldi.layers.Wav2LogSpec method*), 195
set_extra_state() (*lhotse.features.kaldi.layers.Wav2MFCC method*), 222
set_extra_state() (*lhotse.features.kaldi.layers.Wav2Spec method*), 181
set_extra_state() (*lhotse.features.kaldi.layers.Wav2Win method*), 154
share_memory() (*lhotse.features.kaldi.layers.Wav2FFT method*), 168
share_memory() (*lhotse.features.kaldi.layers.Wav2LogFilterBank method*), 208
share_memory() (*lhotse.features.kaldi.layers.Wav2LogSpec method*), 195
share_memory() (*lhotse.features.kaldi.layers.Wav2MFCC method*), 222
share_memory() (*lhotse.features.kaldi.layers.Wav2Spec method*), 181
share_memory() (*lhotse.features.kaldi.layers.Wav2Win method*), 154
shuffle() (*lhotse.audio.RecordingSet method*), 115
shuffle() (*lhotse.cut.CutSet method*), 312
shuffle() (*lhotse.features.base.FeatureSet method*), 138
shuffle() (*lhotse.supervision.SupervisionSet method*), 129
simple_cuts (*lhotse.cut.CutSet property*), 296
snr (*lhotse.cut.MixTrack attribute*), 282
sort_by_duration() (*lhotse.cut.CutSet method*), 301
sort_like() (*lhotse.cut.CutSet method*), 301
soundfile_load() (*in module lhotse.audio*), 117
source (*lhotse.audio.AudioSource attribute*), 103
sources (*lhotse.audio.Recording attribute*), 105
speaker (*lhotse.supervision.SupervisionSegment attribute*), 121
speakers (*lhotse.cut.CutSet property*), 296
speakers_audio_mask() (*lhotse.cut.Cut method*), 262
speakers_audio_mask() (*lhotse.cut.MixedCut method*), 290
speakers_audio_mask() (*lhotse.cut.MonoCut method*), 271

speakers_audio_mask() (*lhotse.cut.PaddingCut method*), 279
speakers_feature_mask() (*lhotse.cut.Cut method*), 261
speakers_feature_mask() (*lhotse.cut.MixedCut method*), 290
speakers_feature_mask() (*lhotse.cut.MonoCut method*), 271
speakers_feature_mask() (*lhotse.cut.PaddingCut method*), 279
SpecAugment (*class in lhotse.dataset.signal_transforms*), 41
Spectrogram (*class in lhotse.features.spectrogram*), 233
SpectrogramConfig (*class in lhotse.features.spectrogram*), 233
SPEECH_DIR
lhotse-prepare-heroico command line option, 88
speech_synthesis (*in module lhotse.dataset*), 34
sph_info() (*in module lhotse.audio*), 118
split() (*lhotse.audio.RecordingSet method*), 110
split() (*lhotse.cut.Cut method*), 258
split() (*lhotse.cut.CutSet method*), 298
split() (*lhotse.cut.MixedCut method*), 291
split() (*lhotse.cut.MonoCut method*), 271
split() (*lhotse.cut.PaddingCut method*), 280
split() (*lhotse.features.base.FeatureSet method*), 138
split() (*lhotse.supervision.SupervisionSet method*), 126
split_lazy() (*lhotse.audio.RecordingSet method*), 111
split_lazy() (*lhotse.cut.CutSet method*), 298
split_lazy() (*lhotse.features.base.FeatureSet method*), 138
split_lazy() (*lhotse.supervision.SupervisionSet method*), 126
split_parallelize_combine() (*in module lhotse.manipulation*), 317
start (*lhotse.cut.Cut attribute*), 257
start (*lhotse.cut.MixedCut property*), 282
start (*lhotse.cut.MonoCut attribute*), 263
start (*lhotse.cut.PaddingCut property*), 274
start (*lhotse.features.base.Features attribute*), 136
start (*lhotse.supervision.AlignmentItem attribute*), 119
start (*lhotse.supervision.SupervisionSegment attribute*), 121
state_dict() (*lhotse.dataset.signal_transforms.SpecAugment method*), 42
state_dict() (*lhotse.features.kaldi.layers.Wav2FFT method*), 168
state_dict() (*lhotse.features.kaldi.layers.Wav2LogFilterBank method*), 208
state_dict() (*lhotse.features.kaldi.layers.Wav2LogSpec method*), 195
state_dict() (*lhotse.features.kaldi.layers.Wav2MFCC method*), 222
state_dict() (*lhotse.features.kaldi.layers.Wav2Spec method*), 181
state_dict() (*lhotse.features.kaldi.layers.Wav2Win method*), 155
storage_key (*lhotse.features.base.Features attribute*), 136
STORAGE_PATH
lhotse-copy-feats command line option, 52
lhotse-feat-extract-cuts command line option, 70
lhotse-feat-extract-cuts-batch command line option, 71
storage_path (*lhotse.features.base.Features attribute*), 136
storage_path (*lhotse.features.io.ChunkedLilcomHdf5Writer property*), 247
storage_path (*lhotse.features.io.FeaturesWriter property*), 240
storage_path (*lhotse.features.io.KaldiWriter property*), 251
storage_path (*lhotse.features.io.LilcomChunkyWriter property*), 249
storage_path (*lhotse.features.io.LilcomFilesWriter property*), 242
storage_path (*lhotse.features.io.LilcomHdf5Writer property*), 246
storage_path (*lhotse.features.io.LilcomURLWriter property*), 250
storage_path (*lhotse.features.io.MemoryLilcomWriter property*), 252
storage_path (*lhotse.features.io.MemoryRawWriter property*), 253
storage_path (*lhotse.features.io.NumpyFilesWriter property*), 243
storage_path (*lhotse.features.io.NumpyHdf5Writer property*), 245
storage_type (*lhotse.features.base.Features attribute*), 136
store_array() (*lhotse.features.io.ChunkedLilcomHdf5Writer method*), 247
store_array() (*lhotse.features.io.FeaturesWriter method*), 240
store_array() (*lhotse.features.io.KaldiWriter method*), 251
store_array() (*lhotse.features.io.LilcomChunkyWriter method*), 249
store_array() (*lhotse.features.io.LilcomFilesWriter method*), 242
store_array() (*lhotse.features.io.LilcomHdf5Writer method*), 246
store_array() (*lhotse.features.io.LilcomURLWriter method*), 250
store_array() (*lhotse.features.io.MemoryLilcomWriter*

method), 252
store_array() (lhotse.features.io.MemoryRawWriter method), 253
store_array() (lhotse.features.io.NumpyFilesWriter method), 243
store_array() (lhotse.features.io.NumpyHdf5Writer method), 245
store_feature_array() (in module lhotse.features.base), 142
subset() (lhotse.audio.RecordingSet method), 111
subset() (lhotse.cut.CutSet method), 299
subset() (lhotse.features.base.FeatureSet method), 138
subset() (lhotse.supervision.SupervisionSet method), 126
supervision_intervals() (lhotse.dataset.input_strategies.AudioSamples method), 37
supervision_intervals() (lhotse.dataset.input_strategies.BatchIO method), 35
supervision_intervals() (lhotse.dataset.input_strategies.OnTheFlyFeatures method), 38
supervision_intervals() (lhotse.dataset.input_strategies.PrecomputedFeatures method), 36
SUPERVISION_MANIFEST
lhotse-cut-random-mixed command line option, 56
supervision_masks() (lhotse.dataset.input_strategies.AudioSamples method), 37
supervision_masks() (lhotse.dataset.input_strategies.BatchIO method), 36
supervision_masks() (lhotse.dataset.input_strategies.OnTheFlyFeatures method), 38
supervision_masks() (lhotse.dataset.input_strategies.PrecomputedFeatures method), 36
SUPERVISIONS
lhotse-fix command line option, 73
lhotse-kaldi-export command line option, 74
lhotse-validate-pair command line option, 101
supervisions (lhotse.cut.Cut attribute), 257
supervisions (lhotse.cut.MixedCut property), 282
supervisions (lhotse.cut.MonoCut attribute), 263
supervisions (lhotse.cut.PaddingCut property), 274
supervisions_audio_mask() (lhotse.cut.Cut method), 262
supervisions_audio_mask() (lhotse.cut.MixedCut method), 291
supervisions_audio_mask() (lhotse.cut.MonoCut method), 271
supervisions_audio_mask() (lhotse.cut.PaddingCut method), 280
supervisions_feature_mask() (lhotse.cut.Cut method), 262
supervisions_feature_mask() (lhotse.cut.MixedCut method), 291
supervisions_feature_mask() (lhotse.cut.MonoCut method), 272
supervisions_feature_mask() (lhotse.cut.PaddingCut method), 280
SupervisionSegment (class in lhotse.supervision), 119
SupervisionSet (class in lhotse.supervision), 123
suppress_audio_loading_errors() (in module lhotse.audio), 118
symbol (lhotse.supervision.AlignmentItem attribute), 119

T

T_destination (lhotse.features.kaldi.layers.Wav2FFT attribute), 158
T_destination (lhotse.features.kaldi.layers.Wav2LogFilterBank attribute), 198
T_destination (lhotse.features.kaldi.layers.Wav2LogSpec attribute), 185
T_destination (lhotse.features.kaldi.layers.Wav2MFCC attribute), 212
T_destination (lhotse.features.kaldi.layers.Wav2Spec attribute), 171
T_destination (lhotse.features.kaldi.layers.Wav2Win attribute), 145
TARGET_DIR
lhotse-download-adept command line option, 59
lhotse-download-aidatatang-200zh command line option, 60
lhotse-download-aishell command line option, 60
lhotse-download-aishell4 command line option, 60
lhotse-download-ali-meeting command line option, 61
lhotse-download-ami command line option, 61
lhotse-download-cmu-arctic command line option, 62
lhotse-download-cmu-indic command line option, 62
lhotse-download-earnings21 command line option, 62
lhotse-download-gigaspeech command line option, 63

lhotse-download-heroico command line option, 63
lhotse-download-hifitts command line option, 63
lhotse-download-libricss command line option, 64
lhotse-download-librimix command line option, 65
lhotse-download-librispeech command line option, 65
lhotse-download-libritts command line option, 65
lhotse-download-ljspeech command line option, 66
lhotse-download-mtedx command line option, 66
lhotse-download-musan command line option, 66
lhotse-download-rir-noise command line option, 66
lhotse-download-spgispeech command line option, 67
lhotse-download-tedlium command line option, 67
lhotse-download-timit command line option, 67
lhotse-download-vctk command line option, 68
lhotse-download-voxceleb1 command line option, 68
lhotse-download-voxceleb2 command line option, 68
lhotse-download-yesno command line option, 69
TEDLIUM_DIR
lhotse-prepare-tedlium command line option, 96
text (*lhotse.supervision.SupervisionSegment attribute*), 121
to() (*lhotse.features.kaldi.layers.Wav2FFT method*), 168
to() (*lhotse.features.kaldi.layers.Wav2LogFilterBank method*), 209
to() (*lhotse.features.kaldi.layers.Wav2LogSpec method*), 195
to() (*lhotse.features.kaldi.layers.Wav2MFCC method*), 222
to() (*lhotse.features.kaldi.layers.Wav2Spec method*), 182
to() (*lhotse.features.kaldi.layers.Wav2Win method*), 155
to_cut() (*lhotse.audio.Recording method*), 107
to_dict() (*lhotse.audio.AudioSource method*), 104
to_dict() (*lhotse.audio.Recording method*), 107
to_dict() (*lhotse.cut.Cut method*), 258
to_dict() (*lhotse.cut.MixedCut method*), 291
to_dict() (*lhotse.cut.MonoCut method*), 272
to_dict() (*lhotse.cut.PaddingCut method*), 280
to_dict() (*lhotse.features.base.FeatureExtractor method*), 133
to_dict() (*lhotse.features.base.Features method*), 137
to_dict() (*lhotse.features.base.TorchaudioFeatureExtractor method*), 136
to_dict() (*lhotse.features.fbank.TorchaudioFbank method*), 229
to_dict() (*lhotse.features.fbank.TorchaudioFbankConfig method*), 226
to_dict() (*lhotse.features.librosa_fbank.LibrosaFbank method*), 239
to_dict() (*lhotse.features.librosa_fbank.LibrosaFbankConfig method*), 236
to_dict() (*lhotse.features.mfcc.TorchaudioMfcc method*), 232
to_dict() (*lhotse.features.mfcc.TorchaudioMfccConfig method*), 230
to_dict() (*lhotse.features.spectrogram.Spectrogram method*), 236
to_dict() (*lhotse.features.spectrogram.SpectrogramConfig method*), 233
to_dict() (*lhotse.supervision.SupervisionSegment method*), 123
to_dicts() (*lhotse.audio.RecordingSet method*), 110
to_dicts() (*lhotse.cut.CutSet method*), 297
to_dicts() (*lhotse.features.base.FeatureSet method*), 137
to_dicts() (*lhotse.supervision.SupervisionSet method*), 125
to_eager() (*lhotse.audio.RecordingSet method*), 115
to_eager() (*lhotse.cut.CutSet method*), 312
to_eager() (*lhotse.features.base.FeatureSet method*), 137
to_eager() (*lhotse.supervision.SupervisionSet method*), 129
to_empty() (*lhotse.features.kaldi.layers.Wav2FFT method*), 170
to_empty() (*lhotse.features.kaldi.layers.Wav2LogFilterBank method*), 210
to_empty() (*lhotse.features.kaldi.layers.Wav2LogSpec method*), 196
to_empty() (*lhotse.features.kaldi.layers.Wav2MFCC method*), 224
to_empty() (*lhotse.features.kaldi.layers.Wav2Spec method*), 183
to_empty() (*lhotse.features.kaldi.layers.Wav2Win method*), 156
to_file() (*lhotse.audio.RecordingSet method*), 115
to_file() (*lhotse.cut.CutSet method*), 312
to_file() (*lhotse.dataset.signal_transforms.GlobalMVN method*), 41
to_file() (*lhotse.features.base.FeatureSet method*),

141
to_file() (*lhotse.supervision.SupervisionSet method*), 129
to_json() (*lhotse.audio.RecordingSet method*), 115
to_json() (*lhotse.cut.CutSet method*), 312
to_json() (*lhotse.features.base.FeatureSet method*), 142
to_json() (*lhotse.supervision.SupervisionSet method*), 130
to_jsonl() (*lhotse.audio.RecordingSet method*), 115
to_jsonl() (*lhotse.cut.CutSet method*), 312
to_jsonl() (*lhotse.features.base.FeatureSet method*), 142
to_jsonl() (*lhotse.supervision.SupervisionSet method*), 130
to_manifest() (*in module lhotse.manipulation*), 318
to_yaml() (*lhotse.audio.RecordingSet method*), 115
to_yaml() (*lhotse.cut.CutSet method*), 313
to_yaml() (*lhotse.features.base.FeatureExtractor method*), 133
to_yaml() (*lhotse.features.base.FeatureSet method*), 142
to_yaml() (*lhotse.features.base.TorchaudioFeatureExtractor method*), 136
to_yaml() (*lhotse.features.fbank.TorchaudioFbank method*), 229
to_yaml() (*lhotse.features.librosa_fbank.LibrosaFbank method*), 239
to_yaml() (*lhotse.features.mfcc.TorchaudioMfcc method*), 233
to_yaml() (*lhotse.features.spectrogram.Spectrogram method*), 236
to_yaml() (*lhotse.supervision.SupervisionSet method*), 130
TokenCollater (*class in lhotse.dataset.collation*), 44
torchaudio_info() (*in module lhotse.audio*), 117
torchaudio_load() (*in module lhotse.audio*), 117
TorchaudioFbank (*class in lhotse.features.fbank*), 226
TorchaudioFbankConfig (*class in lhotse.features.fbank*), 226
TorchaudioFeatureExtractor (*class in lhotse.features.base*), 133
TorchaudioMfcc (*class in lhotse.features.mfcc*), 230
TorchaudioMfccConfig (*class in lhotse.features.mfcc*), 229
tracks (*lhotse.cut.MixedCut attribute*), 282
train() (*lhotse.features.kaldi.layers.Wav2FFT method*), 170
train() (*lhotse.features.kaldi.layers.Wav2LogFilterBank method*), 210
train() (*lhotse.features.kaldi.layers.Wav2LogSpec method*), 197
train() (*lhotse.features.kaldi.layers.Wav2MFCC method*), 224
train() (*lhotse.features.kaldi.layers.Wav2Spec method*), 183
train() (*lhotse.features.kaldi.layers.Wav2Win method*), 156
training (*lhotse.dataset.signal_transforms.DereverbWPE attribute*), 44
training (*lhotse.dataset.signal_transforms.GlobalMVN attribute*), 41
training (*lhotse.dataset.signal_transforms.RandomizedSmoothing attribute*), 43
training (*lhotse.dataset.signal_transforms.SpecAugment attribute*), 43
training (*lhotse.features.kaldi.layers.Wav2FFT attribute*), 171
training (*lhotse.features.kaldi.layers.Wav2LogFilterBank attribute*), 211
training (*lhotse.features.kaldi.layers.Wav2LogSpec attribute*), 198
training (*lhotse.features.kaldi.layers.Wav2MFCC attribute*), 225
training (*lhotse.features.kaldi.layers.Wav2Spec attribute*), 184
training (*lhotse.features.kaldi.layers.Wav2Win attribute*), 157
TRANSCRIPT_DIR
  lhotse-prepare-broadcast-news command line option, 78
  lhotse-prepare-callhome-egyptian command line option, 80
  lhotse-prepare-fisher-spanish command line option, 86
  lhotse-prepare-heroico command line option, 88
transform() (*lhotse.supervision.AlignmentItem method*), 119
transform_alignment() (*lhotse.supervision.SupervisionSegment method*), 123
transform_alignment() (*lhotse.supervision.SupervisionSet method*), 127
transform_text() (*lhotse.cut.CutSet method*), 310
transform_text() (*lhotse.supervision.SupervisionSegment method*), 123
transform_text() (*lhotse.supervision.SupervisionSet method*), 126
transforms (*lhotse.audio.Recording attribute*), 105
trim() (*lhotse.supervision.AlignmentItem method*), 119
trim() (*lhotse.supervision.SupervisionSegment method*), 122
trim_to_supervisions() (*lhotse.cut.Cut method*), 259
trim_to_supervisions() (*lhotse.cut.CutSet method*), 300

trim_to_supervisions() (*lhotse.cut.MixedCut method*), 291
trim_to_supervisions() (*lhotse.cut.MonoCut method*), 272
trim_to_supervisions() (*lhotse.cut.PaddingCut method*), 280
trim_to_unsupervised_segments() (*lhotse.cut.CutSet method*), 301
trimmed_supervisions (*lhotse.cut.Cut property*), 258
trimmed_supervisions (*lhotse.cut.MixedCut property*), 292
trimmed_supervisions (*lhotse.cut.MonoCut property*), 273
trimmed_supervisions (*lhotse.cut.PaddingCut property*), 281
truncate (*lhotse.cut.Cut attribute*), 258
truncate() (*lhotse.cut.CutSet method*), 302
truncate() (*lhotse.cut.MixedCut method*), 283
truncate() (*lhotse.cut.MonoCut method*), 265
truncate() (*lhotse.cut.PaddingCut method*), 274
type (*lhotse.audio.AudioSource attribute*), 103
type (*lhotse.features.base.Features attribute*), 136
type() (*lhotse.features.kaldi.layers.Wav2FFT method*), 170
type() (*lhotse.features.kaldi.layers.Wav2LogFilterBank method*), 210
type() (*lhotse.features.kaldi.layers.Wav2LogSpec method*), 197
type() (*lhotse.features.kaldi.layers.Wav2MFCC method*), 224
type() (*lhotse.features.kaldi.layers.Wav2Spec method*), 183
type() (*lhotse.features.kaldi.layers.Wav2Win method*), 157

U

unmixed_audio (*lhotse.audio.AudioMixer property*), 116
unmixed_feats (*lhotse.features.mixer.FeatureMixer property*), 254
UnsupervisedDataset (*class in lhotse.dataset.unsupervised*), 32
UnsupervisedWaveformDataset (*class in lhotse.dataset.unsupervised*), 32
URL
    lhotse-feat-upload command line option, 72
use_energy (*lhotse.features.fbank.TorchaudioFbankConfig attribute*), 226
use_energy (*lhotse.features.mfcc.TorchaudioMfccConfig attribute*), 230

V

VadDataset (*class in lhotse.dataset.vad*), 35
validate() (*lhotse.dataset.source_separation.DynamicallyMixedSourceSe method*), 34
validate_for_asr() (*in module lhotse.dataset.speech_recognition*), 34
vtln_high (*lhotse.features.fbank.TorchaudioFbankConfig attribute*), 226
vtln_high (*lhotse.features.mfcc.TorchaudioMfccConfig attribute*), 230
vtln_low (*lhotse.features.fbank.TorchaudioFbankConfig attribute*), 226
vtln_low (*lhotse.features.mfcc.TorchaudioMfccConfig attribute*), 230
vtln_warp (*lhotse.features.fbank.TorchaudioFbankConfig attribute*), 226
vtln_warp (*lhotse.features.mfcc.TorchaudioMfccConfig attribute*), 230

W

Wav2FFT (*class in lhotse.features.kaldi.layers*), 157
Wav2LogFilterBank (*class in lhotse.features.kaldi.layers*), 198
Wav2LogSpec (*class in lhotse.features.kaldi.layers*), 184
Wav2MFCC (*class in lhotse.features.kaldi.layers*), 211
Wav2Spec (*class in lhotse.features.kaldi.layers*), 171
Wav2Win (*class in lhotse.features.kaldi.layers*), 144
win_length (*lhotse.features.librosa_fbank.LibrosaFbankConfig attribute*), 236
window (*lhotse.features.librosa_fbank.LibrosaFbankConfig attribute*), 236
window_type (*lhotse.features.fbank.TorchaudioFbankConfig attribute*), 226
window_type (*lhotse.features.kaldi.layers.Wav2FFT property*), 158
window_type (*lhotse.features.kaldi.layers.Wav2LogFilterBank property*), 211
window_type (*lhotse.features.kaldi.layers.Wav2LogSpec property*), 197
window_type (*lhotse.features.kaldi.layers.Wav2MFCC property*), 224
window_type (*lhotse.features.kaldi.layers.Wav2Spec property*), 184
window_type (*lhotse.features.mfcc.TorchaudioMfccConfig attribute*), 229
window_type (*lhotse.features.spectrogram.SpectrogramConfig attribute*), 233
with_alignment_from_ctm() (*lhotse.supervision.SupervisionSet method*), 125
with_features_path_prefix (*lhotse.cut.Cut attribute*), 258
with_features_path_prefix() (*lhotse.cut.CutSet method*), 309
with_features_path_prefix() (*lhotse.cut.MixedCut method*), 288

with_features_path_prefix() (*lhotse.cut.MonoCut method*), 269
with_features_path_prefix() (*lhotse.cut.PaddingCut method*), 277
with_id() (*lhotse.cut.Cut method*), 262
with_id() (*lhotse.cut.MixedCut method*), 293
with_id() (*lhotse.cut.MonoCut method*), 273
with_id() (*lhotse.cut.PaddingCut method*), 281
with_offset() (*lhotse.supervision.AlignmentItem method*), 119
with_offset() (*lhotse.supervision.SupervisionSegment method*), 121
with_path_prefix() (*lhotse.audio.AudioSource method*), 104
with_path_prefix() (*lhotse.audio.Recording method*), 107
with_path_prefix() (*lhotse.audio.RecordingSet method*), 111
with_path_prefix() (*lhotse.features.base.Features method*), 137
with_path_prefix() (*lhotse.features.base.FeatureSet method*), 138
with_recording_path_prefix (*lhotse.cut.Cut attribute*), 258
with_recording_path_prefix() (*lhotse.cut.CutSet method*), 309
with_recording_path_prefix() (*lhotse.cut.MixedCut method*), 288
with_recording_path_prefix() (*lhotse.cut.MonoCut method*), 269
with_recording_path_prefix() (*lhotse.cut.PaddingCut method*), 277
with_traceback() (*lhotse.audio.AudioLoadingError method*), 118
with_traceback() (*lhotse.audio.DurationMismatchError method*), 118
write() (*lhotse.features.io.ChunkedLilcomHdf5Writer method*), 247
write() (*lhotse.features.io.FeaturesWriter method*), 240
write() (*lhotse.features.io.KaldiWriter method*), 251
write() (*lhotse.features.io.LilcomChunkyWriter method*), 249
write() (*lhotse.features.io.LilcomFilesWriter method*), 242
write() (*lhotse.features.io.LilcomHdf5Writer method*), 246
write() (*lhotse.features.io.LilcomURLWriter method*), 250
write() (*lhotse.features.io.MemoryLilcomWriter method*), 252
write() (*lhotse.features.io.MemoryRawWriter method*), 253
write() (*lhotse.features.io.NumpyFilesWriter method*), 243
write() (*lhotse.features.io.NumpyHdf5Writer method*), 245
write_alignment_to_ctm() (*lhotse.supervision.SupervisionSet method*), 125
WSPECIFIER
lhotse-cut-export-to-webdataset command line option, 54

X

xpu() (*lhotse.features.kaldi.layers.Wav2FFT method*), 170
xpu() (*lhotse.features.kaldi.layers.Wav2LogFilterBank method*), 211
xpu() (*lhotse.features.kaldi.layers.Wav2LogSpec method*), 197
xpu() (*lhotse.features.kaldi.layers.Wav2MFCC method*), 224
xpu() (*lhotse.features.kaldi.layers.Wav2Spec method*), 184
xpu() (*lhotse.features.kaldi.layers.Wav2Win method*), 157

Z

zero_grad() (*lhotse.features.kaldi.layers.Wav2FFT method*), 171
zero_grad() (*lhotse.features.kaldi.layers.Wav2LogFilterBank method*), 211
zero_grad() (*lhotse.features.kaldi.layers.Wav2LogSpec method*), 197
zero_grad() (*lhotse.features.kaldi.layers.Wav2MFCC method*), 225
zero_grad() (*lhotse.features.kaldi.layers.Wav2Spec method*), 184
zero_grad() (*lhotse.features.kaldi.layers.Wav2Win method*), 157