Package ‘torchaudio’

February 8, 2023

Title R Interface to 'pytorch''s 'torchaudio'

Version 0.3.1

Description Provides access to datasets, models and processing facilities for deep learning in audio.

License MIT + file LICENSE

Encoding UTF-8

RoxygenNote 7.2.3

Suggests testthat, tuneR, knitr, rmarkdown, stringr, numbers, purrr, scales, httr, viridis

Imports torch (>= 0.3.0), av, fs, rlang, utils, tools, glue, methods

VignetteBuilder knitr

NeedsCompilation no

Author Sigrid Keydana [aut, cre],
Athos Damiani [aut],
Daniel Falbel [aut]

Maintainer Sigrid Keydana <sigrid@posit.co>

Repository CRAN

Date/Publication 2023-02-08 08:40:02 UTC

R topics documented:

`cmuarctic_dataset` *CMU Arctic Dataset*

Description

Create a Dataset for CMU_ARCTIC.

Usage

```
cmuarctic_dataset(
  root,
  url = "aew",
  folder_in_archive = "ARCTIC",
  download = FALSE
)
```

Arguments

- `root` (str): Path to the directory where the dataset is found or downloaded.
- `url` (str, optional): The URL to download the dataset from or the type of the dataset to dowload. (default: `"aew"`) Allowed type values are `"aew"`, `"ahw"`, `"aup"`, `"awb"`, `"axb"`, `"bdl"`, `"clb"`, `"eey"`, `"fem"`, `"gka"`, `"jmk"`, `"ksp"`, `"ljm"`, `"lnh"`, `"rms"`, `"rxr"`, `"slp"` or `"slt"`.
- `folder_in_archive` (str, optional): The top-level directory of the dataset. (default: `"ARCTIC"`)
- `download` (bool, optional): Whether to download the dataset if it is not found at root path. (default: `FALSE`).

Value

a torch::dataset()

`extract_archive` *Extract Archive*

Description

Extract Archive

Usage

```
extract_archive(from_path, to_path = NULL, overwrite = FALSE)
```

Arguments

- `from_path` (str): the path of the archive.
- `to_path` (str, optional): the root path of the extraced files (directory of from_path) (Default: `NULL`)
- `overwrite` (bool, optional): overwrite existing files (Default: `FALSE`)

Value

list: List of paths to extracted files even if not overwritten.

Examples

```
if(torch::torch_is_installed()) {
url = 'http://www.quest.dcs.shef.ac.uk/wmt16_files_mmt/validation.tar.gz'
d <- fs::dir_create(tempdir(), "torchaudio")
from_path <- fs::path(d, basename(url))
utils::download.file(url = url, destfile = from_path)
torchaudio::extract_archive (from_path, d)
}
```

functional_add_noise_shaping

Noise Shaping (functional)

Description

Noise shaping is calculated by error: error[n] = dithered[n] - original[n] noise_shaped_waveform[n] = dithered[n] + error[n-1]

Usage

```
functional_add_noise_shaping(dithered_waveform, waveform)
```

Arguments

`dithered_waveform`	(Tensor) dithered
`waveform`	(Tensor) original

Value

`tensor` of the noise shaped waveform

functional_allpass_biquad

All-pass Biquad Filter (functional)

Description

Design two-pole all-pass filter. Similar to SoX implementation.

Usage

```
functional_allpass_biquad(waveform, sample_rate, central_freq, Q = 0.707)
```

Arguments

`waveform`	(Tensor): audio waveform of dimension of `(..., time)`
`sample_rate`	(int): sampling rate of the waveform, e.g. 44100 (Hz)
`central_freq`	(float): central frequency (in Hz)
`Q`	(float, optional): https://en.wikipedia.org/wiki/Q_factor (Default: `0.707`)

Value

`tensor`: Waveform of dimension of `(..., time)`

References

- https://sox.sourceforge.net/sox.html
- https://webaudio.github.io/Audio-EQ-Cookbook/audio-eq-cookbook.html

---

functional_amplitude_to_db

Amplitude to DB (functional)

Description

Turn a tensor from the power/amplitude scale to the decibel scale.

Usage

```
functional_amplitude_to_db(x, multiplier, amin, db_multiplier, top_db = NULL)
```

Arguments

`x`	(Tensor): Input tensor before being converted to decibel scale
`multiplier`	(float): Use 10.0 for power and 20.0 for amplitude (Default: `10.0`)
`amin`	(float): Number to clamp `x` (Default: `1e-10`)
`db_multiplier`	(float): Log10(max(ref_value and amin))
`top_db`	(float or NULL, optional): Minimum negative cut-off in decibels. A reasonable number is 80. (Default: `NULL`)

Details

This output depends on the maximum value in the input tensor, and so may return different values for an audio clip split into snippets vs. a a full clip.

Value

`tensor`: Output tensor in decibel scale

functional_angle — *Angle (functional)*

Description

Compute the angle of complex tensor input.

Usage

```
functional_angle(complex_tensor)
```

Arguments

`complex_tensor`	(Tensor): Tensor shape of `(..., complex=2)`

Value

`tensor`: Angle of a complex tensor. Shape of `(..., )`

functional_apply_probability_distribution — *Probability Distribution Apply (functional)*

Description

Apply a probability distribution function on a waveform.

Usage

```
functional_apply_probability_distribution(waveform, density_function = "TPDF")
```

Arguments

`waveform`	(Tensor): Tensor of audio of dimension (..., time)
`density_function`	(str, optional): The density function of a continuous random variable (Default: `"TPDF"`) Options: Triangular Probability Density Function - `TPDF` Rectangular Probability Density Function - `RPDF` Gaussian Probability Density Function - `GPDF`

Details

- **Triangular** probability density function (TPDF) dither noise has a triangular distribution; values in the center of the range have a higher probability of occurring.
- **Rectangular** probability density function (RPDF) dither noise has a uniform distribution; any value in the specified range has the same probability of occurring.
- **Gaussian** probability density function (GPDF) has a normal distribution. The relationship of probabilities of results follows a bell-shaped, or Gaussian curve, typical of dither generated by analog sources.

Value

`tensor`: waveform dithered with TPDF

---

functional_bandpass_biquad

Band-pass Biquad Filter (functional)

Description

Design two-pole band-pass filter. Similar to SoX implementation.

Usage

```
functional_bandpass_biquad(
  waveform,
  sample_rate,
  central_freq,
  Q = 0.707,
  const_skirt_gain = FALSE
)
```

Arguments

`waveform`	(Tensor): audio waveform of dimension of `(..., time)`
`sample_rate`	(int): sampling rate of the waveform, e.g. 44100 (Hz)
`central_freq`	(float): central frequency (in Hz)
`Q`	(float, optional): `https://en.wikipedia.org/wiki/Q_factor` (Default: `0.707`)
`const_skirt_gain`	(bool, optional) : If `TRUE`, uses a constant skirt gain (peak gain = Q). If `FALSE`, uses a constant 0dB peak gain. (Default: `FALSE`)

Value

Tensor: Waveform of dimension of `(..., time)`

References

- https://sox.sourceforge.net/sox.html
- https://webaudio.github.io/Audio-EQ-Cookbook/audio-eq-cookbook.html

---

functional_bandreject_biquad
Band-reject Biquad Filter (functional)

Description

Design two-pole band-reject filter. Similar to SoX implementation.

Usage

```
functional_bandreject_biquad(waveform, sample_rate, central_freq, Q = 0.707)
```

Arguments

`waveform`	(Tensor): audio waveform of dimension of `(..., time)`
`sample_rate`	(int): sampling rate of the waveform, e.g. 44100 (Hz)
`central_freq`	(float): central frequency (in Hz)
`Q`	(float, optional): https://en.wikipedia.org/wiki/Q_factor (Default: `0.707`)

Value

`tensor`: Waveform of dimension of `(..., time)`

References

- https://sox.sourceforge.net/sox.html
- https://webaudio.github.io/Audio-EQ-Cookbook/audio-eq-cookbook.html

`functional_band_biquad` *Two-pole Band Filter (functional)*

Description

Design two-pole band filter. Similar to SoX implementation.

Usage

```
functional_band_biquad(
  waveform,
  sample_rate,
  central_freq,
  Q = 0.707,
  noise = FALSE
)
```

Arguments

`waveform`	(Tensor): audio waveform of dimension of `(..., time)`
`sample_rate`	(int): sampling rate of the waveform, e.g. 44100 (Hz)
`central_freq`	(float): central frequency (in Hz)
`Q`	(float, optional): https://en.wikipedia.org/wiki/Q_factor (Default: `0.707`).
`noise`	(bool, optional) : If `TRUE`, uses the alternate mode for un-pitched audio (e.g. percussion). If `FALSE`, uses mode oriented to pitched audio, i.e. voice, singing, or instrumental music (Default: `FALSE`).

Value

`tensor`: Waveform of dimension of `(..., time)`

References

- https://sox.sourceforge.net/sox.html
- https://webaudio.github.io/Audio-EQ-Cookbook/audio-eq-cookbook.html

---

`functional_bass_biquad`
Bass Tone-control Effect (functional)

Description

Design a bass tone-control effect. Similar to SoX implementation.

Usage

```
functional_bass_biquad(
  waveform,
  sample_rate,
  gain,
  central_freq = 100,
  Q = 0.707
)
```

Arguments

`waveform`	(Tensor): audio waveform of dimension of `(..., time)`
`sample_rate`	(int): sampling rate of the waveform, e.g. 44100 (Hz)
`gain`	(float): desired gain at the boost (or attenuation) in dB.
`central_freq`	(float, optional): central frequency (in Hz). (Default: `100`)
`Q`	(float, optional): `https://en.wikipedia.org/wiki/Q_factor` (Default: `0.707`).

Value

`tensor`: Waveform of dimension of `(..., time)`

References

- `https://sox.sourceforge.net/sox.html`
- `https://webaudio.github.io/Audio-EQ-Cookbook/audio-eq-cookbook.html`

functional_biquad *Biquad Filter (functional)*

Description

Perform a biquad filter of input tensor. Initial conditions set to 0. https://en.wikipedia.org/wiki/Digital_biquad_filter

Usage

```
functional_biquad(waveform, b0, b1, b2, a0, a1, a2)
```

Arguments

waveform	(Tensor): audio waveform of dimension of `(..., time)`
b0	(float): numerator coefficient of current input, x[n]
b1	(float): numerator coefficient of input one time step ago x[n-1]
b2	(float): numerator coefficient of input two time steps ago x[n-2]
a0	(float): denominator coefficient of current output y[n], typically 1
a1	(float): denominator coefficient of current output y[n-1]
a2	(float): denominator coefficient of current output y[n-2]

Value

`tensor`: Waveform with dimension of `(..., time)`

functional_complex_norm *Complex Norm (functional)*

Description

Compute the norm of complex tensor input.

Usage

```
functional_complex_norm(complex_tensor, power = 1)
```

Arguments

complex_tensor	(tensor): Tensor shape of `(..., complex=2)`
power	(numeric): Power of the norm. (Default: `1.0`).

Value

`tensor`: Power of the normed input tensor. Shape of `(..., )` functional_compute_deltas

Delta Coefficients (functional)

Description

Compute delta coefficients of a tensor, usually a spectrogram.

Usage

```
functional_compute_deltas(specgram, win_length = 5, mode = "replicate")
```

Arguments

specgram	(Tensor): Tensor of audio of dimension (..., freq, time)
win_length	(int, optional): The window length used for computing delta (Default: 5)
mode	(str, optional): Mode parameter passed to padding (Default: "replicate")

Details

math:

$$d_t = \frac{\sum_{n=1}^{N} n(c_{t+n} - c_{t-n})}{2\sum_{n=1}^{N} n^2}$$

where d_t is the deltas at time t, c_t is the spectrogram coeffcients at time t, N is (win_length-1) %/% 2.

Value

tensor: Tensor of deltas of dimension (..., freq, time)

Examples

```
if(torch::torch_is_installed()) {
library(torch)
library(torchaudio)
specgram = torch_randn(1, 40, 1000)
delta = functional_compute_deltas(specgram)
delta2 = functional_compute_deltas(delta)
}
```

functional_contrast *Contrast Effect (functional)*

Description

Apply contrast effect. Similar to SoX implementation. Comparable with compression, this effect modifies an audio signal to make it sound louder

Usage

```
functional_contrast(waveform, enhancement_amount = 75)
```

Arguments

`waveform`	(Tensor): audio waveform of dimension of `(..., time)`
`enhancement_amount`	(float): controls the amount of the enhancement Allowed range of values for enhancement_amount : 0-100 Note that enhancement_amount = 0 still gives a significant contrast enhancement

Value

`tensor`: Waveform of dimension of `(..., time)`

References

- https://sox.sourceforge.net/sox.html

functional_create_dct *DCT transformation matrix (functional)*

Description

Create a DCT transformation matrix with shape (`n_mels`, `n_mfcc`), normalized depending on norm. https://en.wikipedia.org/wiki/Discrete_cosine_transform

Usage

```
functional_create_dct(n_mfcc, n_mels, norm = NULL)
```

Arguments

`n_mfcc`	(int): Number of mfc coefficients to retain
`n_mels`	(int): Number of mel filterbanks
`norm`	(chr or NULL): Norm to use (either 'ortho' or NULL)

Value

`tensor`: The transformation matrix, to be right-multiplied to row-wise data of size (`n_mels`, `n_mfcc`).

---

functional_create_fb_matrix

Frequency Bin Conversion Matrix (functional)

Description

Create a frequency bin conversion matrix.

Usage

```
functional_create_fb_matrix(
  n_freqs,
  f_min,
  f_max,
  n_mels,
  sample_rate,
  norm = NULL
)
```

Arguments

`n_freqs`	(int): Number of frequencies to highlight/apply
`f_min`	(float): Minimum frequency (Hz)
`f_max`	(float or NULL): Maximum frequency (Hz). If NULL defaults to sample_rate %/% 2
`n_mels`	(int): Number of mel filterbanks
`sample_rate`	(int): Sample rate of the audio waveform
`norm`	(chr) (Optional): If 'slaney', divide the triangular mel weights by the width of the mel band (area normalization). (Default: `NULL`)

Value

`tensor`: Triangular filter banks (fb matrix) of size (`n_freqs`, `n_mels`) meaning number of frequencies to highlight/apply to x the number of filterbanks. Each column is a filterbank so that assuming there is a matrix A of size (..., `n_freqs`), the applied result would be `A * functional_create_fb_matrix(A.size(-1), ...)`.

functional_db_to_amplitude *DB to Amplitude (functional)*

Description

Turn a tensor from the decibel scale to the power/amplitude scale.

Usage

```
functional_db_to_amplitude(x, ref, power)
```

Arguments

x	(Tensor): Input tensor before being converted to power/amplitude scale.
ref	(float): Reference which the output will be scaled by. (Default: `1.0`)
power	(float): If power equals 1, will compute DB to power. If 0.5, will compute DB to amplitude. (Default: `1.0`)

Value

`tensor`: Output tensor in power/amplitude scale.

functional_dcshift *DC Shift (functional)*

Description

Apply a DC shift to the audio. Similar to SoX implementation. This can be useful to remove a DC offset (caused perhaps by a hardware problem in the recording chain) from the audio

Usage

```
functional_dcshift(waveform, shift, limiter_gain = NULL)
```

Arguments

waveform	(Tensor): audio waveform of dimension of `(..., time)`
shift	(float): indicates the amount to shift the audio Allowed range of values for shift : -2.0 to +2.0
limiter_gain	(float): It is used only on peaks to prevent clipping It should have a value much less than 1 (e.g. 0.05 or 0.02)

Value

`tensor`: Waveform of dimension of `(..., time)`

References

- https://sox.sourceforge.net/sox.html

`functional_deemph_biquad` *ISO 908 CD De-emphasis IIR Filter (functional)*

Description

Apply ISO 908 CD de-emphasis (shelving) IIR filter. Similar to SoX implementation.

Usage

```
functional_deemph_biquad(waveform, sample_rate)
```

Arguments

`waveform`	(Tensor): audio waveform of dimension of `(..., time)`
`sample_rate`	(int): sampling rate of the waveform, Allowed sample rate `44100` or `48000`

Value

Tensor: Waveform of dimension of `(..., time)`

References

- https://sox.sourceforge.net/sox.html
- https://webaudio.github.io/Audio-EQ-Cookbook/audio-eq-cookbook.html

functional_detect_pitch_frequency

Detect Pitch Frequency (functional)

Description

It is implemented using normalized cross-correlation function and median smoothing.

Usage

```
functional_detect_pitch_frequency(
  waveform,
  sample_rate,
  frame_time = 10^(-2),
  win_length = 30,
  freq_low = 85,
  freq_high = 3400
)
```

Arguments

`waveform`	(Tensor): Tensor of audio of dimension (..., freq, time)
`sample_rate`	(int): The sample rate of the waveform (Hz)
`frame_time`	(float, optional): Duration of a frame (Default: `10 ** (-2)`).
`win_length`	(int, optional): The window length for median smoothing (in number of frames) (Default: `30`).
`freq_low`	(int, optional): Lowest frequency that can be detected (Hz) (Default: `85`).
`freq_high`	(int, optional): Highest frequency that can be detected (Hz) (Default: `3400`).

Value

Tensor: Tensor of freq of dimension (..., frame)

functional_dither

Dither (functional)

Description

Dither increases the perceived dynamic range of audio stored at a particular bit-depth by eliminating nonlinear truncation distortion (i.e. adding minimally perceived noise to mask distortion caused by quantization).

Usage

```
functional_dither(waveform, density_function = "TPDF", noise_shaping = FALSE)
```

Arguments

waveform (Tensor): Tensor of audio of dimension (..., time)

density_function (str, optional): The density function of a continuous random variable (Default: "TPDF") Options: Triangular Probability Density Function - TPDF Rectangular Probability Density Function - RPDF Gaussian Probability Density Function - GPDF

noise_shaping (bool, optional): a filtering process that shapes the spectral energy of quantisation error (Default: FALSE)

Value

tensor: waveform dithered

---

functional_equalizer_biquad

Biquad Peaking Equalizer Filter (functional)

Description

Design biquad peaking equalizer filter and perform filtering. Similar to SoX implementation.

Usage

```
functional_equalizer_biquad(
  waveform,
  sample_rate,
  center_freq,
  gain,
  Q = 0.707
)
```

Arguments

waveform (Tensor): audio waveform of dimension of (..., time)

sample_rate (int): sampling rate of the waveform, e.g. 44100 (Hz)

center_freq (float): filter's central frequency

gain (float): desired gain at the boost (or attenuation) in dB

Q (float, optional): https://en.wikipedia.org/wiki/Q_factor (Default: 0.707)

Value

Tensor: Waveform of dimension of (..., time)

`functional_flanger` *Flanger Effect (functional)*

Description

Apply a flanger effect to the audio. Similar to SoX implementation.

Usage

```
functional_flanger(
  waveform,
  sample_rate,
  delay = 0,
  depth = 2,
  regen = 0,
  width = 71,
  speed = 0.5,
  phase = 25,
  modulation = "sinusoidal",
  interpolation = "linear"
)
```

Arguments

`waveform`	(Tensor): audio waveform of dimension of `(..., channel, time)` . Max 4 channels allowed
`sample_rate`	(int): sampling rate of the waveform, e.g. 44100 (Hz)
`delay`	(float): desired delay in milliseconds(ms). Allowed range of values are 0 to 30
`depth`	(float): desired delay depth in milliseconds(ms). Allowed range of values are 0 to 10
`regen`	(float): desired regen(feeback gain) in dB. Allowed range of values are -95 to 95
`width`	(float): desired width(delay gain) in dB. Allowed range of values are 0 to 100
`speed`	(float): modulation speed in Hz. Allowed range of values are 0.1 to 10
`phase`	(float): percentage phase-shift for multi-channel. Allowed range of values are 0 to 100
`modulation`	(str): Use either "sinusoidal" or "triangular" modulation. (Default: `sinusoidal`)
`interpolation`	(str): Use either "linear" or "quadratic" for delay-line interpolation. (Default: `linear`)

Value

`tensor`: Waveform of dimension of `(..., channel, time)`

References

- https://sox.sourceforge.net/sox.html
- Scott Lehman, Effects Explained, https://web.archive.org/web/20051125072557/http://www.harmony-central.com/Effects/effects-explained.html

---

functional_gain *Gain (functional)*

Description

Apply amplification or attenuation to the whole waveform.

Usage

```
functional_gain(waveform, gain_db = 1)
```

Arguments

waveform	(Tensor): Tensor of audio of dimension (..., time).
gain_db	(float, optional) Gain adjustment in decibels (dB) (Default: `1.0`).

Value

`tensor`: the whole waveform amplified by gain_db.

---

functional_griffinlim *Griffin-Lim Transformation (functional)*

Description

Compute waveform from a linear scale magnitude spectrogram using the Griffin-Lim transformation. Implementation ported from `librosa`.

Usage

```
functional_griffinlim(
  specgram,
  window,
  n_fft,
  hop_length,
  win_length,
  power,
  normalized,
  n_iter,
  momentum,
  length,
  rand_init
)
```

Arguments

specgram	(Tensor): A magnitude-only STFT spectrogram of dimension (..., freq, frames) where freq is n_fft %/% 2 + 1.
window	(Tensor): Window tensor that is applied/multiplied to each frame/window
n_fft	(int): Size of FFT, creates n_fft %/% 2 + 1 bins
hop_length	(int): Length of hop between STFT windows.
win_length	(int): Window size.
power	(float): Exponent for the magnitude spectrogram, (must be > 0) e.g., 1 for energy, 2 for power, etc.
normalized	(bool): Whether to normalize by magnitude after stft.
n_iter	(int): Number of iteration for phase recovery process.
momentum	(float): The momentum parameter for fast Griffin-Lim. Setting this to 0 recovers the original Griffin-Lim method. Values near 1 can lead to faster convergence, but above 1 may not converge.
length	(int or NULL): Array length of the expected output.
rand_init	(bool): Initializes phase randomly if TRUE, to zero otherwise.

Value

tensor: waveform of (..., time), where time equals the length parameter if given.

---

functional_highpass_biquad

High-pass Biquad Filter (functional)

Description

Design biquad highpass filter and perform filtering. Similar to SoX implementation.

Usage

```
functional_highpass_biquad(waveform, sample_rate, cutoff_freq, Q = 0.707)
```

Arguments

waveform	(Tensor): audio waveform of dimension of (..., time)
sample_rate	(int): sampling rate of the waveform, e.g. 44100 (Hz)
cutoff_freq	(float): filter cutoff frequency
Q	(float, optional): https://en.wikipedia.org/wiki/Q_factor (Default: 0.707)

Value

tensor: Waveform dimension of (..., time)

`functional_lfilter` *An IIR Filter (functional)*

Description

Perform an IIR filter by evaluating difference equation.

Usage

```
functional_lfilter(waveform, a_coeffs, b_coeffs, clamp = TRUE)
```

Arguments

`waveform`	(Tensor): audio waveform of dimension of `(..., time)`. Must be normalized to -1 to 1.
`a_coeffs`	(Tensor): denominator coefficients of difference equation of dimension of `(n_order + 1)`. Lower delays coefficients are first, e.g. `[a0, a1, a2, ...]`. Must be same size as b_coeffs (pad with 0's as necessary).
`b_coeffs`	(Tensor): numerator coefficients of difference equation of dimension of `(n_order + 1)`. Lower delays coefficients are first, e.g. `[b0, b1, b2, ...]`. Must be same size as a_coeffs (pad with 0's as necessary).
`clamp`	(bool, optional): If `TRUE`, clamp the output signal to be in the range [-1, 1] (Default: `TRUE`)

Value

`tensor`: Waveform with dimension of `(..., time)`.

`functional_lowpass_biquad` *Low-pass Biquad Filter (functional)*

Description

Design biquad lowpass filter and perform filtering. Similar to SoX implementation.

Usage

```
functional_lowpass_biquad(waveform, sample_rate, cutoff_freq, Q = 0.707)
```

Arguments

`waveform`	(torch.Tensor): audio waveform of dimension of `(..., time)`
`sample_rate`	(int): sampling rate of the waveform, e.g. 44100 (Hz)
`cutoff_freq`	(float): filter cutoff frequency
`Q`	(float, optional): https://en.wikipedia.org/wiki/Q_factor (Default: `0.707`)

Value

`tensor`: Waveform of dimension of `(..., time)`

`functional_magphase` *Magnitude and Phase (functional)*

Description

Separate a complex-valued spectrogram with shape `(.., 2)` into its magnitude and phase.

Usage

```
functional_magphase(complex_tensor, power = 1)
```

Arguments

`complex_tensor`	(Tensor): Tensor shape of `(.., complex=2)`
`power`	(float): Power of the norm. (Default: `1.0`)

Value

list(`tensor`, `tensor`): The magnitude and phase of the complex tensor

---

`functional_mask_along_axis`
Mask Along Axis (functional)

Description

Apply a mask along `axis`. Mask will be applied from indices `[v_0, v_0 + v)`, where `v` is sampled from `uniform (0, mask_param)`, and `v_0` from `uniform(0, max_v - v)`. All examples will have the same mask interval.

Usage

```
functional_mask_along_axis(specgram, mask_param, mask_value, axis)
```

Arguments

`specgram`	(Tensor): Real spectrogram (channel, freq, time)
`mask_param`	(int): Number of columns to be masked will be uniformly sampled from `[0, mask_param]`
`mask_value`	(float): Value to assign to the masked columns
`axis`	(int): Axis to apply masking on (2 -> frequency, 3 -> time)

Value

Tensor: Masked spectrogram of dimensions (channel, freq, time)

functional_mask_along_axis_iid *Mask Along Axis IID (functional)*

Description

Apply a mask along `axis`. Mask will be applied from indices `[v_0, v_0 + v)`, where `v` is sampled from `uniform (0, mask_param)`, and `v_0` from `uniform(0, max_v - v)`.

Usage

```
functional_mask_along_axis_iid(specgrams, mask_param, mask_value, axis)
```

Arguments

`specgrams`	(Tensor): Real spectrograms (batch, channel, freq, time)
`mask_param`	(int): Number of columns to be masked will be uniformly sampled from `[0, mask_param]`
`mask_value`	(float): Value to assign to the masked columns
`axis`	(int): Axis to apply masking on (3 -> frequency, 4 -> time)

Value

`tensor`: Masked spectrograms of dimensions (batch, channel, freq, time)

functional_mel_scale *Mel Scale (functional)*

Description

Turn a normal STFT into a mel frequency STFT, using a conversion matrix. This uses triangular filter banks.

Usage

```
functional_mel_scale(
  specgram,
  n_mels = 128,
  sample_rate = 16000,
  f_min = 0,
  f_max = NULL,
  n_stft = NULL
)
```

Arguments

specgram	(Tensor): A spectrogram STFT of dimension (..., freq, time).
n_mels	(int, optional): Number of mel filterbanks. (Default: 128)
sample_rate	(int, optional): Sample rate of audio signal. (Default: 16000)
f_min	(float, optional): Minimum frequency. (Default: 0.)
f_max	(float or NULL, optional): Maximum frequency. (Default: sample_rate %/% 2)
n_stft	(int, optional): Number of bins in STFT. Calculated from first input if NULL is given. See n_fft in :class:Spectrogram. (Default: NULL)

Value

tensor: Mel frequency spectrogram of size (..., n_mels, time).

---

functional_mu_law_decoding
Mu Law Decoding (functional)

Description

Decode mu-law encoded signal. For more info see the Wikipedia Entry

Usage

```
functional_mu_law_decoding(x_mu, quantization_channels)
```

Arguments

x_mu	(Tensor): Input tensor
quantization_channels	(int): Number of channels

Details

This expects an input with values between 0 and quantization_channels - 1 and returns a signal scaled between -1 and 1.

Value

tensor: Input after mu-law decoding

functional_mu_law_encoding
Mu Law Encoding (functional)

Description

Encode signal based on mu-law companding. For more info see the Wikipedia Entry

Usage

```
functional_mu_law_encoding(x, quantization_channels)
```

Arguments

`x`	(Tensor): Input tensor
`quantization_channels`	(int): Number of channels

Details

This algorithm assumes the signal has been scaled to between -1 and 1 and returns a signal encoded with values from 0 to quantization_channels - 1.

Value

`tensor`: Input after mu-law encoding

---

functional_overdrive *Overdrive Effect (functional)*

Description

Apply a overdrive effect to the audio. Similar to SoX implementation. This effect applies a non linear distortion to the audio signal.

Usage

```
functional_overdrive(waveform, gain = 20, colour = 20)
```

Arguments

`waveform`	(Tensor): audio waveform of dimension of `(..., time)`
`gain`	(float): desired gain at the boost (or attenuation) in dB Allowed range of values are 0 to 100
`colour`	(float): controls the amount of even harmonic content in the over-driven output. Allowed range of values are 0 to 100

Value

Tensor: Waveform of dimension of `(..., time)`

References

- https://sox.sourceforge.net/sox.html

`functional_phaser` *Phasing Effect (functional)*

Description

Apply a phasing effect to the audio. Similar to SoX implementation.

Usage

```
functional_phaser(
  waveform,
  sample_rate,
  gain_in = 0.4,
  gain_out = 0.74,
  delay_ms = 3,
  decay = 0.4,
  mod_speed = 0.5,
  sinusoidal = TRUE
)
```

Arguments

`waveform`	(Tensor): audio waveform of dimension of `(..., time)`
`sample_rate`	(int): sampling rate of the waveform, e.g. 44100 (Hz)
`gain_in`	(float): desired input gain at the boost (or attenuation) in dB. Allowed range of values are 0 to 1
`gain_out`	(float): desired output gain at the boost (or attenuation) in dB. Allowed range of values are 0 to 1e9
`delay_ms`	(float): desired delay in milli seconds. Allowed range of values are 0 to 5.0
`decay`	(float): desired decay relative to gain-in. Allowed range of values are 0 to 0.99
`mod_speed`	(float): modulation speed in Hz. Allowed range of values are 0.1 to 2
`sinusoidal`	(bool): If `TRUE`, uses sinusoidal modulation (preferable for multiple instruments). If `FALSE`, uses triangular modulation (gives single instruments a sharper phasing effect) (Default: `TRUE`)

Value

`tensor`: Waveform of dimension of `(..., time)`

References

- https://sox.sourceforge.net/sox.html

---

functional_phase_vocoder

Phase Vocoder

Description

Given a STFT tensor, speed up in time without modifying pitch by a factor of `rate`.

Usage

```
functional_phase_vocoder(complex_specgrams, rate, phase_advance)
```

Arguments

`complex_specgrams`	(Tensor): Dimension of `(..., freq, time, complex=2)`
`rate`	(float): Speed-up factor
`phase_advance`	(Tensor): Expected phase advance in each bin. Dimension of (freq, 1)

Value

`tensor`: Complex Specgrams Stretch with dimension of `(..., freq, ceiling(time/rate), complex=2)`

Examples

```
if(torch::torch_is_installed()) {
library(torch)
library(torchaudio)

freq = 1025
hop_length = 512

#  (channel, freq, time, complex=2)
complex_specgrams = torch_randn(2, freq, 300, 2)
rate = 1.3 # Speed up by 30%
phase_advance = torch_linspace(0, pi * hop_length, freq)[.., NULL]
x = functional_phase_vocoder(complex_specgrams, rate, phase_advance)
x$shape # with 231 == ceil (300 / 1.3)
# torch.Size ([2, 1025, 231, 2])
}
```

functional_riaa_biquad

RIAA Vinyl Playback Equalisation (functional)

Description

Apply RIAA vinyl playback equalisation. Similar to SoX implementation.

Usage

```
functional_riaa_biquad(waveform, sample_rate)
```

Arguments

`waveform`	(Tensor): audio waveform of dimension of `(..., time)`
`sample_rate`	(int): sampling rate of the waveform, e.g. 44100 (Hz). Allowed sample rates in Hz : `44100,48000,88200,96000`

Value

`tensor`: Waveform of dimension of `(..., time)`

References

- https://sox.sourceforge.net/sox.html
- https://webaudio.github.io/Audio-EQ-Cookbook/audio-eq-cookbook.html

functional_sliding_window_cmn

sliding-window Cepstral Mean Normalization (functional)

Description

Apply sliding-window cepstral mean (and optionally variance) normalization per utterance.

Usage

```
functional_sliding_window_cmn(
  waveform,
  cmn_window = 600,
  min_cmn_window = 100,
  center = FALSE,
  norm_vars = FALSE
)
```

Arguments

waveform	(Tensor): Tensor of audio of dimension (..., freq, time)
cmn_window	(int, optional): Window in frames for running average CMN computation (int, default = 600)
min_cmn_window	(int, optional): Minimum CMN window used at start of decoding (adds latency only at start). Only applicable if center == FALSE, ignored if center==TRUE (int, default = 100)
center	(bool, optional): If TRUE, use a window centered on the current frame (to the extent possible, modulo end effects). If FALSE, window is to the left. (bool, default = FALSE)
norm_vars	(bool, optional): If TRUE, normalize variance to one. (bool, default = FALSE)

Value

tensor: Tensor of freq of dimension (..., frame)

---

functional_spectrogram
Spectrogram (functional)

Description

Create a spectrogram or a batch of spectrograms from a raw audio signal. The spectrogram can be either magnitude-only or complex.

Usage

```
functional_spectrogram(
  waveform,
  pad,
  window,
  n_fft,
  hop_length,
  win_length,
  power,
  normalized
)
```

Arguments

waveform	(tensor): Tensor of audio of dimension (..., time)
pad	(integer): Two sided padding of signal
window	(tensor or function): Window tensor that is applied/multiplied to each frame/window or a function that generates the window tensor.
n_fft	(integer): Size of FFT

hop_length	(integer): Length of hop between STFT windows
win_length	(integer): Window size
power	(numeric): Exponent for the magnitude spectrogram, (must be > 0) e.g., 1 for energy, 2 for power, etc. If NULL, then the complex spectrum is returned instead.
normalized	(logical): Whether to normalize by magnitude after stft

Value

`tensor`: Dimension (..., freq, time), freq is n_fft %/% 2 + 1 and n_fft is the number of Fourier bins, and time is the number of window hops (n_frame).

`functional_treble_biquad` *Treble Tone-control Effect (functional)*

Description

Design a treble tone-control effect. Similar to SoX implementation.

Usage

```
functional_treble_biquad(
  waveform,
  sample_rate,
  gain,
  central_freq = 3000,
  Q = 0.707
)
```

Arguments

waveform	(Tensor): audio waveform of dimension of `(..., time)`
sample_rate	(int): sampling rate of the waveform, e.g. 44100 (Hz)
gain	(float): desired gain at the boost (or attenuation) in dB.
central_freq	(float, optional): central frequency (in Hz). (Default: `3000`)
Q	(float, optional): https://en.wikipedia.org/wiki/Q_factor (Default: `0.707`).

Value

`tensor`: Waveform of dimension of `(..., time)`

References

- https://sox.sourceforge.net/sox.html
- https://webaudio.github.io/Audio-EQ-Cookbook/audio-eq-cookbook.html

functional_vad *Voice Activity Detector (functional)*

Description

Voice Activity Detector. Similar to SoX implementation. Attempts to trim silence and quiet background sounds from the ends of recordings of speech. The algorithm currently uses a simple cepstral power measurement to detect voice, so may be fooled by other things, especially music.

Usage

```
functional_vad(
  waveform,
  sample_rate,
  trigger_level = 7,
  trigger_time = 0.25,
  search_time = 1,
  allowed_gap = 0.25,
  pre_trigger_time = 0,
  boot_time = 0.35,
  noise_up_time = 0.1,
  noise_down_time = 0.01,
  noise_reduction_amount = 1.35,
  measure_freq = 20,
  measure_duration = NULL,
  measure_smooth_time = 0.4,
  hp_filter_freq = 50,
  lp_filter_freq = 6000,
  hp_lifter_freq = 150,
  lp_lifter_freq = 2000
)
```

Arguments

waveform	(Tensor): Tensor of audio of dimension `(..., time)`
sample_rate	(int): Sample rate of audio signal.
trigger_level	(float, optional): The measurement level used to trigger activity detection. This may need to be cahnged depending on the noise level, signal level, and other characteristics of the input audio. (Default: 7.0)
trigger_time	(float, optional): The time constant (in seconds) used to help ignore short bursts of sound. (Default: 0.25)
search_time	(float, optional): The amount of audio (in seconds) to search for quieter/shorter bursts of audio to include prior to the detected trigger point. (Default: 1.0)
allowed_gap	(float, optional): The allowed gap (in seconds) between quiteter/shorter bursts of audio to include prior to the detected trigger point. (Default: 0.25)

`pre_trigger_time`
: (float, optional): The amount of audio (in seconds) to preserve before the trigger point and any found quieter/shorter bursts. (Default: 0.0)

`boot_time`
: (float, optional) The algorithm (internally) uses adaptive noise estimation/reduction in order to detect the start of the wanted audio. This option sets the time for the initial noise estimate. (Default: 0.35)

`noise_up_time`
: (float, optional) Time constant used by the adaptive noise estimator for when the noise level is increasing. (Default: 0.1)

`noise_down_time`
: (float, optional) Time constant used by the adaptive noise estimator for when the noise level is decreasing. (Default: 0.01)

`noise_reduction_amount`
: (float, optional) Amount of noise reduction to use in the detection algorithm (e.g. 0, 0.5, ...). (Default: 1.35)

`measure_freq`
: (float, optional) Frequency of the algorithm's processing/measurements. (Default: 20.0)

`measure_duration`
: (float, optional) Measurement duration. (Default: Twice the measurement period; i.e. with overlap.)

`measure_smooth_time`
: (float, optional) Time constant used to smooth spectral measurements. (Default: 0.4)

`hp_filter_freq`
: (float, optional) "Brick-wall" frequency of high-pass filter applied at the input to the detector algorithm. (Default: 50.0)

`lp_filter_freq`
: (float, optional) "Brick-wall" frequency of low-pass filter applied at the input to the detector algorithm. (Default: 6000.0)

`hp_lifter_freq`
: (float, optional) "Brick-wall" frequency of high-pass lifter used in the detector algorithm. (Default: 150.0)

`lp_lifter_freq`
: (float, optional) "Brick-wall" frequency of low-pass lifter used in the detector algorithm. (Default: 2000.0)

Details

The effect can trim only from the front of the audio, so in order to trim from the back, the reverse effect must also be used.

Value

Tensor: Tensor of audio of dimension (..., time).

References

- https://sox.sourceforge.net/sox.html

---

functional__combine_max
Combine Max (functional)

Description

Take value from first if bigger than a multiplicative factor of the second, elementwise.

Usage

```
functional__combine_max(a, b, thresh = 0.99)
```

Arguments

`a`	(list(tensor, tensor))
`b`	(list(tensor, tensor))
`thresh`	(float) Default: 0.99

Value

`list(tensor, tensor)`: a list with values tensor and indices tensor.

---

functional__compute_nccf
Normalized Cross-Correlation Function (functional)

Description

Compute Normalized Cross-Correlation Function (NCCF).

Usage

```
functional__compute_nccf(waveform, sample_rate, frame_time, freq_low)
```

Arguments

`waveform`	(Tensor): Tensor of audio of dimension (..., time)
`sample_rate`	(int): sampling rate of the waveform, e.g. 44100 (Hz)
`frame_time`	(float)
`freq_low`	(float)

Value

`tensor` of nccf“

---

functional__find_max_per_frame
Find Max Per Frame (functional)

Description

For each frame, take the highest value of NCCF, apply centered median smoothing, and convert to frequency.

Usage

```
functional__find_max_per_frame(nccf, sample_rate, freq_high)
```

Arguments

nccf	(tensor): Usually a tensor returned by functional__compute_nccf
sample_rate	(int): sampling rate of the waveform, e.g. 44100 (Hz)
freq_high	(int): Highest frequency that can be detected (Hz) Note: If the max among all the lags is very close to the first half of lags, then the latter is taken.

Value

`tensor` with indices

---

functional__generate_wave_table
Wave Table Generator (functional)

Description

A helper function for phaser. Generates a table with given parameters

Usage

```
functional__generate_wave_table(
  wave_type,
  data_type,
  table_size,
  min,
  max,
  phase,
  device
)
```

Arguments

`wave_type`	(str): 'SINE' or 'TRIANGULAR'
`data_type`	(str): desired data_type (`INT` or `FLOAT`)
`table_size`	(int): desired table size
`min`	(float): desired min value
`max`	(float): desired max value
`phase`	(float): desired phase
`device`	(torch_device): Torch device on which table must be generated

Value

`tensor`: A 1D tensor with wave table values

`functional__median_smoothing`

Median Smoothing (functional)

Description

Apply median smoothing to the 1D tensor over the given window.

Usage

```
functional__median_smoothing(indices, win_length)
```

Arguments

`indices`	(Tensor)
`win_length`	(int)

Value

`tensor`

`kaldi_resample_waveform` *Kaldi's Resample Waveform*

Description

Resamples the waveform at the new frequency.

Usage

```
kaldi_resample_waveform(
  waveform,
  orig_freq,
  new_freq,
  lowpass_filter_width = 6
)
```

Arguments

`waveform`	(Tensor): The input signal of size (c, n)
`orig_freq`	(float): The original frequency of the signal
`new_freq`	(float): The desired frequency
`lowpass_filter_width`	(int, optional): Controls the sharpness of the filter, more == sharper but less efficient. We suggest around 4 to 10 for normal use. (Default: `6`)

Details

This matches Kaldi's OfflineFeatureTpl ResampleWaveform which uses a LinearResample (resample a signal at linearly spaced intervals to upsample/downsample a signal). LinearResample (LR) means that the output signal is at linearly spaced intervals (i.e the output signal has a frequency of `new_freq`). It uses sinc/bandlimited interpolation to upsample/downsample the signal.

Value

Tensor: The waveform at the new frequency

References

- https://ccrma.stanford.edu/~jos/resample/Theory_Ideal_Bandlimited_Interpolation.html
- https://github.com/kaldi-asr/kaldi/blob/master/src/feat/resample.h#L56

`kaldi__get_lr_indices_and_weights` *Linear Resample Indices And Weights*

Description

Based on LinearResample::SetIndexesAndWeights where it retrieves the weights for resampling as well as the indices in which they are valid. LinearResample (LR) means that the output signal is at linearly spaced intervals (i.e the output signal has a frequency of `new_freq`).

Usage

```
kaldi__get_lr_indices_and_weights(
  orig_freq,
  new_freq,
  output_samples_in_unit,
  window_width,
  lowpass_cutoff,
  lowpass_filter_width,
  device,
  dtype
)
```

Arguments

`orig_freq`	(float): The original frequency of the signal
`new_freq`	(float): The desired frequency
`output_samples_in_unit`	(int): The number of output samples in the smallest repeating unit: num_samp_out = new_freq / Gcd (orig_freq, new_freq)
`window_width`	(float): The width of the window which is nonzero
`lowpass_cutoff`	(float): The filter cutoff in Hz. The filter cutoff needs to be less than samp_rate_in_hz/2 and less than samp_rate_out_hz/2.
`lowpass_filter_width`	(int): Controls the sharpness of the filter, more == sharper but less efficient. We suggest around 4 to 10 for normal use.
`device`	(torch_device): Torch device on which output must be generated.
`dtype`	(torch::torch_\<dtype\>): Torch dtype such as torch::torch_float

Details

It uses sinc/bandlimited interpolation to upsample/downsample the signal.

The reason why the same filter is not used for multiple convolutions is because the sinc function could sampled at different points in time. For example, suppose a signal is sampled at the timestamps (seconds) 0 16 32 and we want it to be sampled at the timestamps (seconds) 0 5 10 15 20 25

30 35 at the timestamp of 16, the delta timestamps are 16 11 6 1 4 9 14 19 at the timestamp of 32, the delta timestamps are 32 27 22 17 12 8 2 3

As we can see from deltas, the sinc function is sampled at different points of time assuming the center of the sinc function is at 0, 16, and 32 (the deltas [..., 6, 1, 4, ...] for 16 vs [...., 2, 3, ...] for 32)

Example, one case is when the `orig_freq` and `new_freq` are multiples of each other then there needs to be one filter.

A windowed filter function (i.e. Hanning * sinc) because the ideal case of sinc function has infinite support (non-zero for all values) so instead it is truncated and multiplied by a window function which gives it less-than-perfect rolloff [1].

[1] Chapter 16: Windowed-Sinc Filters, https://www.dspguide.com/ch16/1.htm

Value

Tensor, Tensor): A tuple of `min_input_index` (which is the minimum indices where the window is valid, size (`output_samples_in_unit`)) and `weights` (which is the weights which correspond with min_input_index, size (`output_samples_in_unit`, `max_weight_width`)).

`kaldi__get_num_lr_output_samples` *Linear Resample Output Samples*

Description

Based on LinearResample::GetNumOutputSamples. LinearResample (LR) means that the output signal is at linearly spaced intervals (i.e the output signal has a frequency of `new_freq`). It uses sinc/bandlimited interpolation to upsample/downsample the signal.

Usage

```
kaldi__get_num_lr_output_samples(input_num_samp, samp_rate_in, samp_rate_out)
```

Arguments

`input_num_samp`	(int): The number of samples in the input
`samp_rate_in`	(float): The original frequency of the signal
`samp_rate_out`	(float): The desired frequency

Value

int: The number of output samples

linear_to_mel_frequency

Linear to mel frequency

Description

Converts frequencies from the linear scale to mel scale.

Usage

```
linear_to_mel_frequency(
  frequency_in_hertz,
  mel_break_frequency_hertz = 2595,
  mel_high_frequency_q = 700
)
```

Arguments

`frequency_in_hertz`
: (numeric) tensor of frequencies in hertz to be converted to mel scale.

`mel_break_frequency_hertz`
: (numeric) scalar. (Default to 2595.0)

`mel_high_frequency_q`
: (numeric) scalar. (Default to 700.0)

Value

`tensor`

list_audio_backends *List available audio backends*

Description

List available audio backends

Usage

```
list_audio_backends()
```

Value

character vector with the list of available backends.

mel_to_linear_frequency *Mel to linear frequency*

Description

Converts frequencies from the mel scale to linear scale.

Usage

```
mel_to_linear_frequency(
  frequency_in_mel,
  mel_break_frequency_hertz = 2595,
  mel_high_frequency_q = 700
)
```

Arguments

`frequency_in_mel`
: (numeric) tensor of frequencies in mel to be converted to linear scale.

`mel_break_frequency_hertz`
: (numeric) scalar. (Default to 2595.0)

`mel_high_frequency_q`
: (numeric) scalar. (Default to 700.0)

Value

tensor

model_melresnet *MelResNet*

Description

MelResNet layer uses a stack of ResBlocks on spectrogram. Pass the input through the MelResNet layer.

Usage

```
model_melresnet(
  n_res_block = 10,
  n_freq = 128,
  n_hidden = 128,
  n_output = 128,
  kernel_size = 5
)
```

Arguments

`n_res_block`	the number of ResBlock in stack. (Default: 10)
`n_freq`	the number of bins in a spectrogram. (Default: 128)
`n_hidden`	the number of hidden dimensions of resblock. (Default: 128)
`n_output`	the number of output dimensions of melresnet. (Default: 128)
`kernel_size`	the number of kernel size in the first Conv1d layer. (Default: 5)

Details

forward param: specgram (Tensor): the input sequence to the MelResNet layer (n_batch, n_freq, n_time).

Value

Tensor shape: (n_batch, n_output, n_time - kernel_size + 1)

Examples

```
if(torch::torch_is_installed()) {
 melresnet = model_melresnet()
 input = torch::torch_rand(10, 128, 512)  # a random spectrogram
 output = melresnet(input)  # shape: (10, 128, 508)
}
```

---

model_resblock   *ResBlock*

Description

ResNet block based on "Deep Residual Learning for Image Recognition". Pass the input through the ResBlock layer. The paper link is https://arxiv.org/pdf/1512.03385.pdf.

Usage

```
model_resblock(n_freq = 128)
```

Arguments

`n_freq`	the number of bins in a spectrogram. (Default: 128)

Details

forward param: specgram (Tensor): the input sequence to the ResBlock layer (n_batch, n_freq, n_time).

Value

Tensor shape: (n_batch, n_freq, n_time)

Examples

```
if(torch::torch_is_installed()) {
resblock = model_resblock()
input = torch::torch_rand(10, 128, 512)  # a random spectrogram
output = resblock(input)  # shape: (10, 128, 512)
}
```

---

model_stretch2d *Stretch2d*

Description

Upscale the frequency and time dimensions of a spectrogram. Pass the input through the Stretch2d layer.

Usage

```
model_stretch2d(time_scale, freq_scale)
```

Arguments

time_scale	the scale factor in time dimension
freq_scale	the scale factor in frequency dimension

Details

forward param: specgram (Tensor): the input sequence to the Stretch2d layer (..., n_freq, n_time).

Value

Tensor shape: (..., n_freq * freq_scale, n_time * time_scale)

Examples

```
if(torch::torch_is_installed()) {
 stretch2d = model_stretch2d(time_scale=10, freq_scale=5)

 input = torch::torch_rand(10, 100, 512)  # a random spectrogram
 output = stretch2d(input)  # shape: (10, 500, 5120)
}
```

model_upsample_network

UpsampleNetwork

Description

Upscale the dimensions of a spectrogram. Pass the input through the UpsampleNetwork layer.

Usage

```
model_upsample_network(
  upsample_scales,
  n_res_block = 10,
  n_freq = 128,
  n_hidden = 128,
  n_output = 128,
  kernel_size = 5
)
```

Arguments

`upsample_scales`	the list of upsample scales.
`n_res_block`	the number of ResBlock in stack. (Default: `10`)
`n_freq`	the number of bins in a spectrogram. (Default: `128`)
`n_hidden`	the number of hidden dimensions of resblock. (Default: `128`)
`n_output`	the number of output dimensions of melresnet. (Default: `128`)
`kernel_size`	the number of kernel size in the first Conv1d layer. (Default: `5`)

Details

forward param: specgram (Tensor): the input sequence to the UpsampleNetwork layer (n_batch, n_freq, n_time)

Value

Tensor shape: (n_batch, n_freq, (n_time - kernel_size + 1) * total_scale), (n_batch, n_output, (n_time - kernel_size + 1) * total_scale) where total_scale is the product of all elements in upsample_scales.

Examples

```
if(torch::torch_is_installed()) {
 upsamplenetwork = model_upsample_network(upsample_scales=c(4, 4, 16))
 input = torch::torch_rand (10, 128, 10)  # a random spectrogram
 output = upsamplenetwork (input)  # shape: (10, 1536, 128), (10, 1536, 128)
}
```

model_wavernn *WaveRNN*

Description

WaveRNN model based on the implementation from fatchord. The original implementation was introduced in "Efficient Neural Audio Synthesis". #' Pass the input through the WaveRNN model.

Usage

```
model_wavernn(
  upsample_scales,
  n_classes,
  hop_length,
  n_res_block = 10,
  n_rnn = 512,
  n_fc = 512,
  kernel_size = 5,
  n_freq = 128,
  n_hidden = 128,
  n_output = 128
)
```

Arguments

`upsample_scales`	the list of upsample scales.
`n_classes`	the number of output classes.
`hop_length`	the number of samples between the starts of consecutive frames.
`n_res_block`	the number of ResBlock in stack. (Default: `10`)
`n_rnn`	the dimension of RNN layer. (Default: `512`)
`n_fc`	the dimension of fully connected layer. (Default: `512`)
`kernel_size`	the number of kernel size in the first Conv1d layer. (Default: `5`)
`n_freq`	the number of bins in a spectrogram. (Default: `128`)
`n_hidden`	the number of hidden dimensions of resblock. (Default: `128`)
`n_output`	the number of output dimensions of melresnet. (Default: `128`)

Details

forward param:

waveform the input waveform to the WaveRNN layer (n_batch, 1, (n_time - kernel_size + 1) * hop_length)

specgram the input spectrogram to the WaveRNN layer (n_batch, 1, n_freq, n_time)

The input channels of waveform and spectrogram have to be 1. The product of `upsample_scales` must equal `hop_length`.

Value

Tensor shape: (n_batch, 1, (n_time - kernel_size + 1) * hop_length, n_classes)

Examples

```
if(torch::torch_is_installed()) {
wavernn <- model_wavernn(upsample_scales=c(2,2,3), n_classes=5, hop_length=12)

waveform <- torch::torch_rand(3,1,(10 - 5 + 1)*12)
spectrogram <- torch::torch_rand(3,1,128,10)
# waveform shape:  (n_batch, n_channel, (n_time - kernel_size + 1) * hop_length)
output <- wavernn(waveform, spectrogram)
}
```

---

speechcommand_dataset *Speech Commands Dataset*

Description

Speech Commands Dataset

Usage

```
speechcommand_dataset(
  root,
  url = "speech_commands_v0.02",
  folder_in_archive = "SpeechCommands",
  download = FALSE,
  normalization = NULL
)
```

Arguments

`root`	(str): Path to the directory where the dataset is found or downloaded.
`url`	(str, optional): The URL to download the dataset from, or the type of the dataset to dowload. Allowed type values are `"speech_commands_v0.01"` and `"speech_commands_v0.02"` (default: `"speech_commands_v0.02"`)
`folder_in_archive`	(str, optional): The top-level directory of the dataset. (default: `"SpeechCommands"`)
`download`	(bool, optional): Whether to download the dataset if it is not found at root path. (default: `FALSE`).
`normalization`	(NULL, bool, int or function): Optional normalization. If boolean TRUE, then output is divided by 2^31. Assuming the input is signed 32-bit audio, this normalizes to [-1, 1]. If numeric, then output is divided by that number. If function, then the output is passed as a paramete to the given function, then the output is divided by the result. (Default: NULL)

Value

a torch::dataset()

---

torchaudio_info *Audio Information*

Description

Retrieve audio metadata.

Usage

```
torchaudio_info(filepath)
```

Arguments

`filepath`	(str) path to the audio file.

Value

AudioMetaData: an R6 class with fields sample_rate, channels, samples.

Examples

```
path <- system.file("waves_yesno/1_1_0_1_1_0_1_1.wav", package = "torchaudio")
torchaudio_info(path)
```

---

torchaudio_load *Load Audio File*

Description

Loads an audio file from disk using the default loader (getOption("torchaudio.loader")).

Usage

```
torchaudio_load(
  filepath,
  offset = 0L,
  duration = -1L,
  unit = c("samples", "time")
)
```

Arguments

- `filepath` (str): Path to audio file
- `offset` (int): Number of frames (or seconds) from the start of the file to begin data loading. (Default: `0`)
- `duration` (int): Number of frames (or seconds) to load. `-1` to load everything after the offset. (Default: `-1`)
- `unit` (str): "sample" or "time". If "sample" duration and offset will be interpreted as frames, and as seconds otherwise.

`transform_amplitude_to_db` *Amplitude to DB*

Description

Turn a tensor from the power/amplitude scale to the decibel scale.

Usage

```
transform_amplitude_to_db(stype = "power", top_db = NULL)
```

Arguments

- `stype` (str, optional): scale of input tensor ('power' or 'magnitude'). The power being the elementwise square of the magnitude. (Default: `'power'`)
- `top_db` (float or NULL, optional): Minimum negative cut-off in decibels. A reasonable number is 80. (Default: `NULL`)

Details

This output depends on the maximum value in the input tensor, and so may return different values for an audio clip split into snippets vs. a a full clip.

forward param: x (Tensor): Input tensor before being converted to decibel scale

Value

`tensor`: Output tensor in decibel scale `transform_complex_norm` *Complex Norm*

Description

Compute the norm of complex tensor input.

Usage

```
transform_complex_norm(power = 1)
```

Arguments

`power`	(float, optional): Power of the norm. (Default: to `1.0`)

Details

forward param: complex_tensor (Tensor): Tensor shape of `(..., complex=2)`.

Value

Tensor: norm of the input tensor, shape of `(..., )`.

`transform_compute_deltas` *Delta Coefficients*

Description

Compute delta coefficients of a tensor, usually a spectrogram.

Usage

```
transform_compute_deltas(win_length = 5, mode = "replicate")
```

Arguments

`win_length`	(int): The window length used for computing delta. (Default: `5`)
`mode`	(str): Mode parameter passed to padding. (Default: `'replicate'`)

Details

forward param: specgram (Tensor): Tensor of audio of dimension (..., freq, time).

See functional_compute_deltas for more details.

Value

Tensor: Tensor of deltas of dimension (..., freq, time).

---

transform_fade *Fade In/Out*

Description

Add a fade in and/or fade out to an waveform.

Usage

```
transform_fade(fade_in_len = 0, fade_out_len = 0, fade_shape = "linear")
```

Arguments

fade_in_len	(int, optional): Length of fade-in (time frames). (Default: 0)
fade_out_len	(int, optional): Length of fade-out (time frames). (Default: 0)
fade_shape	(str, optional): Shape of fade. Must be one of: "quarter_sine", "half_sine", "linear", "logarithmic", "exponential". (Default: "linear")

Details

forward param: waveform (Tensor): Tensor of audio of dimension (..., time).

Value

Tensor: Tensor of audio of dimension (..., time).

---

transform_frequencymasking
Frequency-domain Masking

Description

Apply masking to a spectrogram in the frequency domain.

Usage

```
transform_frequencymasking(freq_mask_param, iid_masks)
```

Arguments

freq_mask_param
: (int): maximum possible length of the mask. Indices uniformly sampled from [0, freq_mask_param).

iid_masks
: (bool, optional): whether to apply different masks to each example/channel in the batch. (Default: `FALSE`) This option is applicable only when the input tensor is 4D.

Value

not implemented yet.

---

transform_inverse_mel_scale

Inverse Mel Scale

Description

Solve for a normal STFT from a mel frequency STFT, using a conversion matrix. This uses triangular filter banks.

Usage

```
transform_inverse_mel_scale(
  n_stft,
  n_mels = 128,
  sample_rate = 16000,
  f_min = 0,
  f_max = NULL,
  max_iter = 1e+05,
  tolerance_loss = 1e-05,
  tolerance_change = 1e-08,
  ...
)
```

Arguments

n_stft	(int): Number of bins in STFT. See `n_fft` in transform_spectrogram.
n_mels	(int, optional): Number of mel filterbanks. (Default: `128`)
sample_rate	(int, optional): Sample rate of audio signal. (Default: `16000`)
f_min	(float, optional): Minimum frequency. (Default: `0.`)
f_max	(float or NULL, optional): Maximum frequency. (Default: `sample_rate %/% 2`)
max_iter	(int, optional): Maximum number of optimization iterations. (Default: `100000`)
tolerance_loss	(float, optional): Value of loss to stop optimization at. (Default: `1e-5`)

tolerance_change
(float, optional): Difference in losses to stop optimization at. (Default: `1e-8`)

... (optional): Arguments passed to the SGD optimizer. Argument lr will default to 0.1 if not specied.(Default: `NULL`)

Details

forward param: melspec (Tensor): A Mel frequency spectrogram of dimension (..., `n_mels`, time)

It minimizes the euclidian norm between the input mel-spectrogram and the product between the estimated spectrogram and the filter banks using SGD.

Value

Tensor: Linear scale spectrogram of size (..., freq, time)

`transform_mel_scale` *Mel Scale*

Description

Turn a normal STFT into a mel frequency STFT, using a conversion matrix. This uses triangular filter banks.

Usage

```
transform_mel_scale(
  n_mels = 128,
  sample_rate = 16000,
  f_min = 0,
  f_max = NULL,
  n_stft = NULL
)
```

Arguments

n_mels	(int, optional): Number of mel filterbanks. (Default: `128`)
sample_rate	(int, optional): Sample rate of audio signal. (Default: `16000`)
f_min	(float, optional): Minimum frequency. (Default: `0.`)
f_max	(float or NULL, optional): Maximum frequency. (Default: `sample_rate // 2`)
n_stft	(int, optional): Number of bins in STFT. Calculated from first input if NULL is given. See `n_fft` in :class:`Spectrogram`. (Default: `NULL`)

Details

forward param: specgram (Tensor): Tensor of audio of dimension (..., freq, time).

Value

`tensor`: Mel frequency spectrogram of size (..., `n_mels`, time).

`transform_mel_spectrogram`
Mel Spectrogram

Description

Create MelSpectrogram for a raw audio signal. This is a composition of Spectrogram and MelScale.

Usage

```
transform_mel_spectrogram(
  sample_rate = 16000,
  n_fft = 400,
  win_length = NULL,
  hop_length = NULL,
  f_min = 0,
  f_max = NULL,
  pad = 0,
  n_mels = 128,
  window_fn = torch::torch_hann_window,
  power = 2,
  normalized = FALSE,
  ...
)
```

Arguments

`sample_rate`	(int, optional): Sample rate of audio signal. (Default: `16000`)
`n_fft`	(int, optional): Size of FFT, creates `n_fft // 2 + 1` bins. (Default: `400`)
`win_length`	(int or NULL, optional): Window size. (Default: `n_fft`)
`hop_length`	(int or NULL, optional): Length of hop between STFT windows. (Default: `win_length // 2`)
`f_min`	(float, optional): Minimum frequency. (Default: `0.`)
`f_max`	(float or NULL, optional): Maximum frequency. (Default: `NULL`)
`pad`	(int, optional): Two sided padding of signal. (Default: `0`)
`n_mels`	(int, optional): Number of mel filterbanks. (Default: `128`)
`window_fn`	(function, optional): A function to create a window tensor that is applied/multiplied to each frame/window. (Default: `torch_hann_window`)
`power`	(float, optional): Power of the norm. (Default: to `2.0`)
`normalized`	(logical): Whether to normalize by magnitude after stft (Default: `FALSE`)
`...`	(optional): Arguments for window function.

Details

forward param: waveform (Tensor): Tensor of audio of dimension (..., time).

Value

`tensor`: Mel frequency spectrogram of size (..., `n_mels`, time).

Sources

- https://gist.github.com/kastnerkyle/179d6e9a88202ab0a2fe
- https://timsainb.github.io/spectrograms-mfccs-and-inversion-in-python.html
- https://haythamfayek.com/2016/04/21/speech-processing-for-machine-learning.html

Examples

```
#'   Example
## Not run:

if(torch::torch_is_installed()) {
mp3_path <- system.file("sample_audio_1.mp3", package = "torchaudio")
sample_mp3 <- transform_to_tensor(tuneR_loader(mp3_path))
# (channel, n_mels, time)
mel_specgram <- transform_mel_spectrogram(sample_rate = sample_mp3[[2]])(sample_mp3[[1]])
}

## End(Not run)
```

`transform_mfcc` *Mel-frequency Cepstrum Coefficients*

Description

Create the Mel-frequency cepstrum coefficients from an audio signal.

Usage

```
transform_mfcc(
  sample_rate = 16000,
  n_mfcc = 40,
  dct_type = 2,
  norm = "ortho",
  log_mels = FALSE,
  ...
)
```

Arguments

- `sample_rate` (int, optional): Sample rate of audio signal. (Default: `16000`)
- `n_mfcc` (int, optional): Number of mfc coefficients to retain. (Default: `40`)
- `dct_type` (int, optional): type of DCT (discrete cosine transform) to use. (Default: `2`)
- `norm` (str, optional): norm to use. (Default: `'ortho'`)
- `log_mels` (bool, optional): whether to use log-mel spectrograms instead of db-scaled. (Default: `FALSE`)
- `...` (optional): arguments for transform_mel_spectrogram.

Details

forward param: waveform (tensor): Tensor of audio of dimension (..., time)

By default, this calculates the MFCC on the DB-scaled Mel spectrogram. This output depends on the maximum value in the input spectrogram, and so may return different values for an audio clip split into snippets vs. a a full clip.

Value

`tensor`: specgram_mel_db of size (..., `n_mfcc`, time).

---

transform_mu_law_decoding

Mu Law Decoding

Description

Decode mu-law encoded signal. For more info see the Wikipedia Entry

Usage

```
transform_mu_law_decoding(quantization_channels = 256)
```

Arguments

- `quantization_channels` (int, optional): Number of channels. (Default: `256`)

Details

This expects an input with values between 0 and quantization_channels - 1 and returns a signal scaled between -1 and 1.

forward param: x_mu (Tensor): A mu-law encoded signal which needs to be decoded.

Value

Tensor: The signal decoded.

`transform_mu_law_encoding` *Mu Law Encoding*

Description

Encode signal based on mu-law companding. For more info see the Wikipedia Entry

Usage

```
transform_mu_law_encoding(quantization_channels = 256)
```

Arguments

`quantization_channels`
: (int, optional): Number of channels. (Default: `256`)

Details

forward param: x (Tensor): A signal to be encoded.

This algorithm assumes the signal has been scaled to between -1 and 1 and returns a signal encoded with values from 0 to quantization_channels - 1.

Value

x_mu (Tensor): An encoded signal.

`transform_resample` *Signal Resample*

Description

Resample a signal from one frequency to another. A resampling method can be given.

Usage

```
transform_resample(
  orig_freq = 16000,
  new_freq = 16000,
  resampling_method = "sinc_interpolation"
)
```

Arguments

- `orig_freq` (float, optional): The original frequency of the signal. (Default: `16000`)
- `new_freq` (float, optional): The desired frequency. (Default: `16000`)
- `resampling_method` (str, optional): The resampling method. (Default: `'sinc_interpolation'`)

Details

forward param: waveform (Tensor): Tensor of audio of dimension (..., time).

Value

Tensor: Output signal of dimension (..., time).

---

transform_sliding_window_cmn

sliding-window Cepstral Mean Normalization

Description

Apply sliding-window cepstral mean (and optionally variance) normalization per utterance.

Usage

```
transform_sliding_window_cmn(
  cmn_window = 600,
  min_cmn_window = 100,
  center = FALSE,
  norm_vars = FALSE
)
```

Arguments

- `cmn_window` (int, optional): Window in frames for running average CMN computation (int, default = 600)
- `min_cmn_window` (int, optional): Minimum CMN window used at start of decoding (adds latency only at start). Only applicable if center == `FALSE`, ignored if center==`TRUE` (int, default = 100)
- `center` (bool, optional): If `TRUE`, use a window centered on the current frame (to the extent possible, modulo end effects). If `FALSE`, window is to the left. (bool, default = `FALSE`)
- `norm_vars` (bool, optional): If `TRUE`, normalize variance to one. (bool, default = `FALSE`)

Details

forward param: waveform (Tensor): Tensor of audio of dimension (..., time).

Value

Tensor: Tensor of audio of dimension (..., time).

---

transform_spectrogram *Spectrogram*

Description

Create a spectrogram or a batch of spectrograms from a raw audio signal. The spectrogram can be either magnitude-only or complex.

Usage

```
transform_spectrogram(
  n_fft = 400,
  win_length = NULL,
  hop_length = NULL,
  pad = 0L,
  window_fn = torch::torch_hann_window,
  power = 2,
  normalized = FALSE,
  ...
)
```

Arguments

`n_fft`	(integer): Size of FFT
`win_length`	(integer): Window size
`hop_length`	(integer): Length of hop between STFT windows
`pad`	(integer): Two sided padding of signal
`window_fn`	(tensor or function): Window tensor that is applied/multiplied to each frame/window or a function that generates the window tensor.
`power`	(numeric): Exponent for the magnitude spectrogram, (must be > 0) e.g., 1 for energy, 2 for power, etc. If NULL, then the complex spectrum is returned instead.
`normalized`	(logical): Whether to normalize by magnitude after stft
`...`	(optional) Arguments for window function.

Details

forward param: waveform (tensor): Tensor of audio of dimension (..., time)

Value

tensor: Dimension (..., freq, time), freq is n_fft %/% 2 + 1 and n_fft is the number of Fourier bins, and time is the number of window hops (n_frame).

transform_timemasking *Time-domain Masking*

Description

Apply masking to a spectrogram in the time domain.

Usage

```
transform_timemasking(time_mask_param, iid_masks)
```

Arguments

time_mask_param	(int): maximum possible length of the mask. Indices uniformly sampled from [0, time_mask_param).
iid_masks	(bool, optional): whether to apply different masks to each example/channel in the batch. (Default: `FALSE`) This option is applicable only when the input tensor is 4D.

Value

not implemented yet.

transform_time_stretch *Time Stretch*

Description

Stretch stft in time without modifying pitch for a given rate.

Usage

```
transform_time_stretch(hop_length = NULL, n_freq = 201, fixed_rate = NULL)
```

Arguments

hop_length	(int or NULL, optional): Length of hop between STFT windows. (Default: `win_length // 2`)
n_freq	(int, optional): number of filter banks from stft. (Default: `201`)
fixed_rate	(float or NULL, optional): rate to speed up or slow down by. If NULL is provided, rate must be passed to the forward method. (Default: `NULL`)

Details

forward param: complex_specgrams (Tensor): complex spectrogram (..., freq, time, complex=2).

overriding_rate (float or NULL, optional): speed up to apply to this batch. If no rate is passed, use `self$fixed_rate`. (Default: `NULL`)

Value

Tensor: Stretched complex spectrogram of dimension (..., freq, ceil(time/rate), complex=2).

`transform_to_tensor` *Convert an audio object into a tensor*

Description

Converts a numeric vector, as delivered by the backend, into a `torch_tensor` of shape (channels x samples). If provided by the backend, attributes "channels" and "sample_rate" will be used.

Usage

```
transform_to_tensor(
  audio,
  out = NULL,
  normalization = TRUE,
  channels_first = TRUE
)
```

Arguments

`audio`	(numeric): A numeric vector, as delivered by the backend.
`out`	(Tensor): An optional output tensor to use instead of creating one. (Default: `NULL`)
`normalization`	(bool, float or function): Optional normalization. If boolean `TRUE`, then output is divided by `2^(bits-1)`. If `bits` info is not available it assumes the input is signed 32-bit audio. If `numeric`, then output is divided by that number. If `function`, then the output is passed as a parameter to the given function, then the output is divided by the result. (Default: `TRUE`)
`channels_first`	(bool): Set channels first or length first in result. (Default: `TRUE`)

Value

```
list(Tensor, int), containing
- the audio content, encoded as `[C x L]` or `[L x C]` where L is the number of audio frames and
    C is the number of channels
- the sample rate of the audio (as listed in the metadata of the file)
```

transform_vad *Voice Activity Detector*

Description

Voice Activity Detector. Similar to SoX implementation.

Usage

```
transform_vad(
  sample_rate,
  trigger_level = 7,
  trigger_time = 0.25,
  search_time = 1,
  allowed_gap = 0.25,
  pre_trigger_time = 0,
  boot_time = 0.35,
  noise_up_time = 0.1,
  noise_down_time = 0.01,
  noise_reduction_amount = 1.35,
  measure_freq = 20,
  measure_duration = NULL,
  measure_smooth_time = 0.4,
  hp_filter_freq = 50,
  lp_filter_freq = 6000,
  hp_lifter_freq = 150,
  lp_lifter_freq = 2000
)
```

Arguments

sample_rate	(int): Sample rate of audio signal.
trigger_level	(float, optional): The measurement level used to trigger activity detection. This may need to be cahnged depending on the noise level, signal level, and other characteristics of the input audio. (Default: 7.0)
trigger_time	(float, optional): The time constant (in seconds) used to help ignore short bursts of sound. (Default: 0.25)
search_time	(float, optional): The amount of audio (in seconds) to search for quieter/shorter bursts of audio to include prior the detected trigger point. (Default: 1.0)
allowed_gap	(float, optional): The allowed gap (in seconds) between quiteter/shorter bursts of audio to include prior to the detected trigger point. (Default: 0.25)
pre_trigger_time	(float, optional): The amount of audio (in seconds) to preserve before the trigger point and any found quieter/shorter bursts. (Default: 0.0)

`boot_time` (float, optional) The algorithm (internally) uses adaptive noise estimation/reduction in order to detect the start of the wanted audio. This option sets the time for the initial noise estimate. (Default: 0.35)

`noise_up_time` (float, optional) Time constant used by the adaptive noise estimator for when the noise level is increasing. (Default: 0.1)

`noise_down_time` (float, optional) Time constant used by the adaptive noise estimator for when the noise level is decreasing. (Default: 0.01)

`noise_reduction_amount` (float, optional) Amount of noise reduction to use in the detection algorithm (e.g. 0, 0.5, ...). (Default: 1.35)

`measure_freq` (float, optional) Frequency of the algorithm's processing/measurements. (Default: 20.0)

`measure_duration` (float, optional) Measurement duration. (Default: Twice the measurement period; i.e. with overlap.)

`measure_smooth_time` (float, optional) Time constant used to smooth spectral measurements. (Default: 0.4)

`hp_filter_freq` (float, optional) "Brick-wall" frequency of high-pass filter applied at the input to the detector algorithm. (Default: 50.0)

`lp_filter_freq` (float, optional) "Brick-wall" frequency of low-pass filter applied at the input to the detector algorithm. (Default: 6000.0)

`hp_lifter_freq` (float, optional) "Brick-wall" frequency of high-pass lifter used in the detector algorithm. (Default: 150.0)

`lp_lifter_freq` (float, optional) "Brick-wall" frequency of low-pass lifter used in the detector algorithm. (Default: 2000.0)

Details

Attempts to trim silence and quiet background sounds from the ends of recordings of speech. The algorithm currently uses a simple cepstral power measurement to detect voice, so may be fooled by other things, especially music.

The effect can trim only from the front of the audio, so in order to trim from the back, the reverse effect must also be used.

forward param: waveform (Tensor): Tensor of audio of dimension `(..., time)`

Value

torch::nn_module()

References

- https://sox.sourceforge.net/sox.html

`transform_vol` *Add a volume to an waveform.*

Description

Add a volume to an waveform.

Usage

```
transform_vol(gain, gain_type = "amplitude")
```

Arguments

`gain`	(float): Interpreted according to the given gain_type: If `gain_type` = `amplitude`, `gain` is a positive amplitude ratio. If `gain_type` = `power`, `gain` is a power (voltage squared). If `gain_type` = `db`, `gain` is in decibels.
`gain_type`	(str, optional): Type of gain. One of: `amplitude`, `power`, `db` (Default: `amplitude`)

Details

forward param: waveform (Tensor): Tensor of audio of dimension (..., time).

Value

Tensor: Tensor of audio of dimension (..., time).

`transform__axismasking` *Axis Masking*

Description

Apply masking to a spectrogram.

Usage

```
transform__axismasking(mask_param, axis, iid_masks)
```

Arguments

`mask_param`	(int): Maximum possible length of the mask.
`axis`	(int): What dimension the mask is applied on.
`iid_masks`	(bool): Applies iid masks to each of the examples in the batch dimension. This option is applicable only when the input tensor is 4D.

Details

forward param: specgram (Tensor): Tensor of dimension (..., freq, time).

mask_value (float): Value to assign to the masked columns.

Value

Tensor: Masked spectrogram of dimensions (..., freq, time).

`yesno_dataset` *YesNo Dataset*

Description

Create a Dataset for YesNo

Usage

```
yesno_dataset(
  root,
  url = "http://www.openslr.org/resources/1/waves_yesno.tar.gz",
  folder_in_archive = "waves_yesno",
  download = FALSE,
  transform = NULL,
  target_transform = NULL
)
```

Arguments

`root` (str): Path to the directory where the dataset is found or downloaded.

`url` (str, optional): The URL to download the dataset from. (default: `"[http://www.openslr.org/resource`

`folder_in_archive` (str, optional): The top-level directory of the dataset. (default: `"waves_yesno"`)

`download` (bool, optional): Whether to download the dataset if it is not found at root path. (default: `FALSE`).

`transform` (callable, optional): Optional transform applied on waveform. (default: `NULL`)

`target_transform` (callable, optional): Optional transform applied on utterance. (default: `NULL`)

Value

tuple: `(waveform, sample_rate, labels)`

Index