Package ‘torchaudio’

May 5, 2021

Title R Interface to ‘pytorch’’s ‘torchaudio’
Version 0.2.0
Description Provides access to datasets, models and preprocessing facilities for deep learning in audio.
License MIT + file LICENSE
Encoding UTF-8
LazyData false
RoxygenNote 7.1.1
Suggests testthat, monitoR, tuneR, knitr, rmarkdown, stringr, numbers, purrr, scales, httr, viridis, av
Imports torch (>= 0.3.0), fs, rlang, rappdirs, utils, tools, glue, Rcpp, withr
VignetteBuilder knitr
LinkingTo Rcpp
NeedsCompilation yes
Author Athos Damiani [aut, cre]
Maintainer Athos Damiani <athos.damiani@gmail.com>
Repository CRAN
Date/Publication 2021-05-05 15:20:07 UTC

R topics documented:

  audiofile_loader .................................................. 3
  av_loader ........................................................... 4
  backend_utils_list_audio_backends ................................ 5
  cmuarcit_dataset .................................................. 5
  extract_archive .................................................... 6
  functional_add_noise_shaping ...................................... 6
  functional_allpass_biquad ......................................... 7
  functional_amplitude_to_db ....................................... 8
  functional_angle .................................................. 8
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>functional_apply_probability_distribution</td>
<td>9</td>
</tr>
<tr>
<td>functional_bandpass_biquad</td>
<td>10</td>
</tr>
<tr>
<td>functional_bandreject_biquad</td>
<td>11</td>
</tr>
<tr>
<td>functional_band_biquad</td>
<td>11</td>
</tr>
<tr>
<td>functional_bass_biquad</td>
<td>12</td>
</tr>
<tr>
<td>functional_biquad</td>
<td>13</td>
</tr>
<tr>
<td>functional_complex_norm</td>
<td>14</td>
</tr>
<tr>
<td>functional_compute_deltas</td>
<td>14</td>
</tr>
<tr>
<td>functional_contrast</td>
<td>15</td>
</tr>
<tr>
<td>functional_create_dct</td>
<td>16</td>
</tr>
<tr>
<td>functional_create_fb_matrix</td>
<td>16</td>
</tr>
<tr>
<td>functional_db_to_amplitude</td>
<td>17</td>
</tr>
<tr>
<td>functional_dcsift</td>
<td>18</td>
</tr>
<tr>
<td>functional_deemph_biquad</td>
<td>18</td>
</tr>
<tr>
<td>functional_detect_pitch_frequency</td>
<td>19</td>
</tr>
<tr>
<td>functional_dither</td>
<td>20</td>
</tr>
<tr>
<td>functional_equalizer_biquad</td>
<td>20</td>
</tr>
<tr>
<td>functional_flanger</td>
<td>21</td>
</tr>
<tr>
<td>functional_gain</td>
<td>22</td>
</tr>
<tr>
<td>functional_griffinlim</td>
<td>23</td>
</tr>
<tr>
<td>functional_highpass_biquad</td>
<td>24</td>
</tr>
<tr>
<td>functional_lfilter</td>
<td>24</td>
</tr>
<tr>
<td>functional_lowpass_biquad</td>
<td>25</td>
</tr>
<tr>
<td>functional_magnphase</td>
<td>25</td>
</tr>
<tr>
<td>functional_mask_along_axis</td>
<td>26</td>
</tr>
<tr>
<td>functional_mask_along_axis_iid</td>
<td>26</td>
</tr>
<tr>
<td>functional_mel_scale</td>
<td>27</td>
</tr>
<tr>
<td>functional_mu_law_decoding</td>
<td>28</td>
</tr>
<tr>
<td>functional_mu_law_encoding</td>
<td>28</td>
</tr>
<tr>
<td>functional_overdrive</td>
<td>29</td>
</tr>
<tr>
<td>functional_phaser</td>
<td>30</td>
</tr>
<tr>
<td>functional_phase_vocoder</td>
<td>31</td>
</tr>
<tr>
<td>functional_riaa_biquad</td>
<td>32</td>
</tr>
<tr>
<td>functional_sliding_window_cmn</td>
<td>32</td>
</tr>
<tr>
<td>functional_spectrogram</td>
<td>33</td>
</tr>
<tr>
<td>functional_treble_biquad</td>
<td>34</td>
</tr>
<tr>
<td>functional_vad</td>
<td>35</td>
</tr>
<tr>
<td>functional__combine_max</td>
<td>37</td>
</tr>
<tr>
<td>functional__compute_nccf</td>
<td>37</td>
</tr>
<tr>
<td>functional__find_max_per_frame</td>
<td>38</td>
</tr>
<tr>
<td>functional__generate_wave_table</td>
<td>38</td>
</tr>
<tr>
<td>functional__median_smoothing</td>
<td>39</td>
</tr>
<tr>
<td>info</td>
<td>40</td>
</tr>
<tr>
<td>kaldiresample_waveform</td>
<td>40</td>
</tr>
<tr>
<td>kaldigetlrindicesandweights</td>
<td>41</td>
</tr>
<tr>
<td>kaldigetnumlroutputsamples</td>
<td>42</td>
</tr>
<tr>
<td>linear_to_mel_frequency</td>
<td>43</td>
</tr>
<tr>
<td>mel_to_linear_frequency</td>
<td>44</td>
</tr>
</tbody>
</table>
Load an audio located at 'filepath' using audiofile backend.
Usage

audiofile_loader(
    filepath,
    offset = 0L,
    duration = 0L,
    unit = c("samples", "time")
)

Arguments

filepath (str) path to the audio file.
offset (num) the sample (or the second if unit = 'time') where the audio should start.
duration (num) how many samples (or how many seconds if unit = 'time') should be extracted.
unit (str) 'samples' or 'time'

Description

Load an audio located at 'filepath' using av package.

Usage

av_loader(filepath, offset = 0L, duration = 0L, unit = c("samples", "time"))

Arguments

filepath (str) path to the audio file.
offset (num) the sample (or the second if unit = 'time') where the audio should start.
duration (num) how many samples (or how many seconds if unit = 'time') should be extracted.
unit (str) 'samples' or 'time'
backend_utils_list_audio_backends

List Available Audio Backends

Description
List Available Audio Backends

Usage
backend_utils_list_audio_backends()

Value
character vector with the list of available backends.

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 download. (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.

Arguments (str, optional): the root path of the extracted 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'
from_path = './validation.tar.gz'
to_path = './'
utils::download.file(url = url, destfile = from_path)
torchaudio::extract_archive (from_path, to_path)
}

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)
**functional_allpass_biquad**

**Arguments**

- `dithered_waveform` (Tensor): dithered waveform
- `original_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)

**Value**

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

**References**

- [https://webaudio.github.io/Audio-EQ-Cookbook/audio-eq-cookbook.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 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)
**functional_apply_probability_distribution**

**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)

**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
**Description**

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

**Usage**

```python
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)
- **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://webaudio.github.io/Audio-EQ-Cookbook/audio-eq-cookbook.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**

```python
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)

**Value**

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

**References**

- [https://webaudio.github.io/Audio-EQ-Cookbook/audio-eq-cookbook.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**

```python
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)
- **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://webaudio.github.io/Audio-EQ-Cookbook/audio-eq-cookbook.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**

```python
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)
**Value**

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

**References**

- [https://webaudio.github.io/Audio-EQ-Cookbook/audio-eq-cookbook.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.  

**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_compute_deltas

Complex Norm (functional)

Description
Compute the norm of complex tensor input.

Usage
functional_compute_deltas(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 coefficients at time \( t \), \( N \) is \( \text{(win_length - 1)/2} \).
*functional_contrast*

**Value**

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

**Examples**

```
if(torch::torch_is_installed()) {
    library(torch)
    library(torchaudio)
    specgram = torch_randn(1, 40, 100)
    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**

**functional_create_dct**  
*DCT transformation matrix (functional)*

### Description
Create a DCT transformation matrix with shape \((n_{mels}, n_{mfcc})\), normalized depending on norm.  

### 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 \((n_{mel}s, n_{mfcc})\).

---

**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,
    n_mels,
    sample_rate,
    norm = NULL
)
```
functional_db_to_amplitude

**Description**

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

**Usage**

```python
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.

---

### 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_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**
```python
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**

---

**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**
```python
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
functional_detect_pitch_frequency

**Value**

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

**References**

- [https://webaudio.github.io/Audio-EQ-Cookbook/audio-eq-cookbook.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**

```python
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**

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

**Arguments**

- `waveform` (Tensor): Tensor of audio of dimension (... time)
- `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**

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

#### Flanger Effect (functional)

#### Description

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

#### Usage

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

#### Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>waveform</td>
<td>(Tensor): audio waveform of dimension of (..., channel, time). Max 4 channels allowed</td>
</tr>
<tr>
<td>sample_rate</td>
<td>(int): sampling rate of the waveform, e.g. 44100 (Hz)</td>
</tr>
<tr>
<td>delay</td>
<td>(float): desired delay in milliseconds (ms). Allowed range of values are 0 to 30</td>
</tr>
<tr>
<td>depth</td>
<td>(float): desired delay depth in milliseconds (ms). Allowed range of values are 0 to 10</td>
</tr>
<tr>
<td>regen</td>
<td>(float): desired regen (feedback gain) in dB. Allowed range of values are -95 to 95</td>
</tr>
<tr>
<td>width</td>
<td>(float): desired width (delay gain) in dB. Allowed range of values are 0 to 100</td>
</tr>
<tr>
<td>speed</td>
<td>(float): modulation speed in Hz. Allowed range of values are 0.1 to 10</td>
</tr>
</tbody>
</table>
functional_gain

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

• http://sox.sourceforge.net/sox.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**

```python
functional_griffinlim(
    specgram,  # Tensor: A magnitude-only STFT spectrogram of dimension (... , freq, frames) where freq is n_fft %/% 2 + 1.
    window,  # 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.
)
```

**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

#### 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)
**functional_lowpass_biquad**

**Value**

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

**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

**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 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 \((0,\text{mask\_param})\), and \(v_0\) from uniform\((0,\text{max\_v} - v)\). All examples will have the same mask interval.

**Usage**

```python
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 \([0, \text{mask\_param}]\)
- `mask_value` (float): Value to assign to the masked columns
- `axis` (int): Axis to apply masking on (3 -> frequency, 4 -> time)

**Value**

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

**Value**

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

---

**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**

```python
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**

```python
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**

functional_phaser  Phasing Effect (functional)

Description

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

Usage

```python
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

**Description**

Given a STFT tensor, speed up in time without modifying pitch by a factor of \( r \).

**Usage**

```python
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**

```python
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**

```python
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://webaudio.github.io/Audio-EQ-Cookbook/audio-eq-cookbook.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**

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

**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**

```python
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
functional_treble_biquad

Description

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

Usage

```r
functional_treble_biquad(
  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 = 3000,  # float, optional: central frequency (in Hz). (Default: 3000)
  Q = 0.707        # float, optional: https://en.wikipedia.org/wiki/Q_factor (Default: 0.707).
)
```

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

References

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

```python
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 changed 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 quieter/shorter bursts of audio to include prior to the detected trigger point. (Default: 0.25)
**functional_vad**

`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**

**Combine Max (functional)**

**Description**

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

**Usage**

```python
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.

---

**Normalized Cross-Correlation Function (functional)**

**Description**

Compute Normalized Cross-Correlation Function (NCCF).

**Usage**

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

**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

---

**Description**

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

**Usage**

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

**Arguments**

- indices (Tensor)
- win_length (int)

**Value**

- tensor
info  

**Audio Information**

**Description**
Retrive metadata from mp3 or wave file without load the audio samples in memory.

**Usage**
info(filepath)

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filepath</td>
<td>(str) path to the mp3/wav file.</td>
</tr>
</tbody>
</table>

**Value**

list(sample_rate, channels, samples)

---

**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**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>waveform</td>
<td>(Tensor): The input signal of size (c, n)</td>
</tr>
<tr>
<td>orig_freq</td>
<td>(float): The original frequency of the signal</td>
</tr>
<tr>
<td>new_freq</td>
<td>(float): The desired frequency</td>
</tr>
<tr>
<td>lowpass_filter_width</td>
<td>(int, optional): Controls the sharpness of the filter, more == sharper but less efficient. We suggest around 4 to 10 for normal use. (Default: 6)</td>
</tr>
</tbody>
</table>
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

---

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

```c
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 times-
\mbox{stamps (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].


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)).
linear_to_mel_frequency

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

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
mel_to_linear_frequency

**Mel to linear frequency**

**Description**

Converts frequencies from the mel scale to linear scale.

**Usage**

```r
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**

```r
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 \((n_{\text{batch}}, n_{\text{freq}}, n_{\text{time}})\).

Value

- Tensor shape: \((n_{\text{batch}}, n_{\text{output}}, n_{\text{time}} - \text{kernel}_\text{size} + 1)\)

Examples

```python
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)
}
```

---

**ResBlock**

Description


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 \((n_{\text{batch}}, n_{\text{freq}}, n_{\text{time}})\).
Value

Tensor shape: (n_batch, n_freq, n_time)

Examples

```cpp
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)
}
```

Description

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

Usage

```cpp
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

```cpp
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

```r
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

```r
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)
}
```
**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**

```python
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`. 
mp3_info 49

**Value**

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

**Examples**

```r
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)
}
```

---

**mp3_info**  MP3 Information

**Description**

Retrieve metadata from mp3 without load the audio samples in memory.

**Usage**

`mp3_info(filepath)`

**Arguments**

filepath  (chr) path to mp3 file

**Value**

AudioMetaData: sample_rate, channels, samples

**Examples**

```r
mp3_path <- system.file("sample_audio_1.mp3", package = "torchaudio")
mp3_info(mp3_path)
```
**set_audio_backend**  
*Set the backend for I/O operation*

**Description**
Set the backend for I/O operation

**Usage**
```
set_audio_backend(backend)
```

**Arguments**
- **backend** (str): one of 'av_loader', 'audiofile_loader' or 'tuneR_loader'.

**Value**
invisible(NULL).
It will set `torchaudio.loader` and `torchaudio.loader.name` options:
```
options( torchaudio.loader = rlang::as_function(backend), torchaudio.loader.name = backend )
```

**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 download. Allowed type values are "speech_commands_v0.01" and "speech_commands_v0.02" (default: "speech_commands_v0.02")
torchaudio_load

**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 parameter to the given function, then the output is divided by the result. (Default: NULL)

**Value**
- a torch::dataset()

---

**torchaudio_load**

*Load Audio File*

**Description**

Loads an audio file from disk into a tensor

**Usage**

```r
torchaudio_load(
  filepath,
  out = NULL,
  normalization = NULL,
  channels_first = TRUE,
  duration = 0L,
  offset = 0L,
  unit = c("sample", "time"),
  signalinfo = NULL,
  encodinginfo = NULL,
  filetype = NULL
)
```

**Arguments**

- **filepath** (str): Path to audio file
- **out** (Tensor): An optional output tensor to use instead of creating one. (Default: NULL)
- **normalization** (NULL, bool, float 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 parameter to the given function, then the output is divided by the result. (Default: NULL)
channels_first (bool): Set channels first or length first in result. (Default: TRUE)
duration (int): Number of frames (or seconds) to load. 0 to load everything after the offset. (Default: 0)
offset (int): Number of frames (or seconds) from the start of the file to begin data loading. (Default: 0)
unit (str): "sample" or "time". If "sample" duration and offset will be interpreted as frames, and as seconds otherwise.
signalinfo (str): A sox_signalinfo_t type, which could be helpful if the audio type cannot be automatically determined. (Default: NULL)
encodinginfo (str): A sox_encodinginfo_t type, which could be set if the audio type cannot be automatically determined. (Default: NULL)
filetype (str): A filetype or extension to be set if sox cannot determine it automatically. (Default: NULL)

Value

list(Tensor, int): An output tensor of size `[C x L)` or `[L x C]` where
L is the number of audio frames and
C is the number of channels.
An integer which is the sample rate of the audio (as listed in the metadata of the file)

Examples

```r
## Not run:
if(torch::torch_is_installed()) {
  mp3_filename <- system.file("sample_audio_2.mp3", package = "torchaudio")
  data = torchaudio_load(mp3_filename)
  print(data[[1]]$size())
  norm_fun <- function(x) torch::torch_abs(x)$max()
  data_vol_normalized = torchaudio_load(mp3_filename, normalization= norm_fun)
  print(data_vol_normalized[[1]]$abs()$max())
}

## End(Not run)
```

--

**torchaudio_loader**  
Load Audio File

**Description**

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

Usage

```r
torchaudio_loader(
    filepath,
    offset = 0L,
    duration = 0L,
    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. 0 to load everything after the offset. (Default: 0)
- **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

```r
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 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**

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

**Arguments**

- `power` *(float, optional)*: Power of the norm. (Default: 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**

```python
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: \texttt{replicate})

**Details**

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

See \texttt{functional_compute_deltas} for more details.
transform_fade

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

**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)
```
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**

```r
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)

**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_mel_scale

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).
transform_mel_spectrogram

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.
transform_mfcc

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://haythamfayek.com/2016/04/21/speech-processing-for-machine-learning.html

Examples

```r
# 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

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

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 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](https://en.wikipedia.org/wiki/Mu-law_encoding).

**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"
)
```
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).
**transform_spectrogram**  

**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**

```r
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_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)

---

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_to_tensor**  

**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 tuneR Wave object or numeric vector into a torch_tensor of shape (Channels x Samples). Convert Audio Object to Tensor.

**Usage**

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

**Arguments**

audio (numeric or Wave): A numeric vector or Wave object, usually from tuneR::readMP3, tuneR::readWave or monitoR::readMP3.

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)

**Details**

If audio is a numeric vector, attributes "channels" and "sample_rate" will be used if exists. Numeric vectors returned from av::read_audio_bin have both attributes by default.
Value

list(Tensor, int): An output tensor of size `[C x L]` or `[L x C]` where
  L is the number of audio frames and
  C is the number of channels.
  An integer which is 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 changed 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)
transform_vad

allowed_gap (float, optional): The allowed gap (in seconds) between quieter/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()
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.
tuneR_loader

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).

tuneR_loader
tuneR_loader

Description
Load an audio located at 'filepath' using tuneR package.

Usage
tuneR_loader(filepath, offset = 0L, duration = 0L, unit = c("samples", "time"))

Arguments
filepath (str) path to the audio file.
offset (num) the sample (or the second if unit = 'time') where the audio should start.
duration (num) how many samples (or how many seconds if unit = 'time') should be extracted.
unit (str) 'samples' or 'time'

wav_info

Wave Information

Description
Retrieve metadata from wav without load the audio samples in memory.

Usage
wav_info(filepath)

Arguments
filepath (chr) path to wav file

Value
AudioMetaData: sample_rate, channels, samples
Examples

```r
wav_path <- system.file("waves_yesno/1_1_0_1_1_0_1_1.wav", package = "torchaudio")
wav_info(wav_path)
```

---

**yesno_dataset**

*YesNo Dataset*

**Description**

Create a Dataset for YesNo

**Usage**

```r
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/resources/1/waves_yesno.tar.gz")
- `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

audiofile_loader, 3
av::read_audio_bin, 65
av_loader, 4
backend_utils_list_audio_backends, 5
cmuarctic_dataset, 5
extract_archive, 6
functional__combine_max, 37
functional__compute_nccf, 37, 38
functional__find_max_per_frame, 38
functional__generate_wave_table, 38
functional__median_smoothing, 39
functional_add_noise_shaping, 6
functional_allpass_biquad, 7
functional_amplitude_to_db, 8
functional_angle, 8
functional_apply_probability_distribution, 9
functional_band_biquad, 11
functional_bandpass_biquad, 10
functional_bandreject_biquad, 11
functional_bass_biquad, 12
functional_biquad, 13
functional_complex_norm, 14
functional_compute_dct, 14, 54
functional_contrast, 15
functional_create_dct, 16
functional_create_fb_matrix, 16
functional_db_to_amplitude, 17
functional_dcshift, 18
functional_deemph_biquad, 18
functional_detect_pitch_frequency, 19
functional_dither, 20
functional_equalizer_biquad, 20
functional_flanger, 21
functional_gain, 22
functional_griffinlim, 23
functional_highpass_biquad, 24
functional_lfilter, 24
functional_lowpass_biquad, 25
functional_magphase, 25
functional_mask_along_axis, 26
functional_mask_along_axis_iid, 26
functional_mel_scale, 27
functional_mu_law_decoding, 28
functional_mu_law_encoding, 28
functional_overdrive, 29
functional_phase_vocoder, 31
functional Phaser, 30
functional_riaa_biquad, 32
functional_sliding_window_cmn, 32
functional_spectrogram, 33
functional_treble_biquad, 34
functional_vad, 35
info, 40
kaldi__get_lr_indices_and_weights, 41
kaldi__get_num_lr_output_samples, 42
kaldi_resample_waveform, 40
linear_to_mel_frequency, 43
mel_to_linear_frequency, 44
model_melresnet, 44
model_resblock, 45
model_stretch2d, 46
model_upsample_network, 47
model_wavernn, 48
monitoR::readMP3, 65
mp3_info, 49
set_audio_backend, 50
speechcommand_dataset, 50
torch::torch_float, 42
torchaudio_load, 51
torchaudio_loader, 52

71
transform_axismasking, 68
transform_amplitude_to_db, 53
transform_complex_norm, 54
transform_compute_deltas, 54
transform_fade, 55
transform_frequencymasking, 55
transform_inverse_mel_scale, 56
transform_mel_scale, 57
transform_mel_spectrogram, 58, 60
transform_mfcc, 59
transform_mu_law_decoding, 60
transform_mu_law_encoding, 61
transform_resample, 61
transform_sliding_window_cmn, 62
transform_spectrogram, 56, 63
transform_time_stretch, 64
transform_timemasking, 64
transform_to_tensor, 65
transform_vad, 66
transform_vol, 68
tuner::readMP3, 65
tuner::readWave, 65
tuner::loader, 69
wav_info, 69
yesno_dataset, 70