torchvideo Documentation

Release 0.0.0

Will Price

Nov 04, 2020

PACKAGE REFERENCE

Similar to `torchvision`, *`torchvideo`* is a library for working with video in pytorch. It contains transforms and dataset classes. It is built atop of `torchvision` and designed to be used in conjunction.

CHAPTER

ONE

TORCHVIDEO.DATASETS

1.1 Datasets

Contents

1.1.1 VideoDataset

class torchvideo.datasets.**VideoDataset**(*root_path*, *label_set=None*, *sampler=FullVideoSampler()*, *transform=None*)

Bases: `torch.utils.data.dataset.Dataset`

Abstract base class that all `VideoDatasets` inherit from. If you are implementing your own `VideoDataset`, you should inherit from this class.

Parameters

- **root_path** (`Union[str, Path]`) – Path to dataset on disk.
- **label_set** (`Optional[LabelSet]`) – Optional label set for labelling examples.
- **sampler** (*`FrameSampler`*) – Optional sampler for drawing frames from each video.
- **transform** (`Optional[Callable[[Any], Tensor]]`) – Optional transform over the list of frames.

__getitem__(*index*)

Load an example by index

Parameters **index** (`int`) – index of the example within the dataset.

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

Returns Example transformed by `transform` if one was passed during instantiation, otherwise the example is converted to a tensor without any transformations applied to it. Additionally, if a label set is present, the method return a tuple: `(video_tensor, label)`

__len__()

Total number of examples in the dataset

Return type int

labels = None
The labels corresponding to the examples in the dataset. To get the label for example at index i you simple call dataset.labels[i], although this will be returned by __getitem__ if this field is not None.

1.1.2 ImageFolderVideoDataset

class torchvideo.datasets.**ImageFolderVideoDataset**(*root_path*, *filename_template*, *filter=None*, *label_set=None*, *sampler=FullVideoSampler()*, *transform=None*, *frame_counter=None*)
Bases: torchvideo.datasets.video_dataset.VideoDataset

Dataset stored as a folder containing folders of images, where each folder represents a video.

The folder hierarchy should look something like this:

```
root/video1/frame_000001.jpg
root/video1/frame_000002.jpg
root/video1/frame_000003.jpg
...

root/video2/frame_000001.jpg
root/video2/frame_000002.jpg
root/video2/frame_000003.jpg
root/video2/frame_000004.jpg
...
```

Parameters

- **root_path** (Union[str, Path]) – Path to dataset on disk. Contents of this folder should be example folders, each with frames named according to the filename_template argument.
- **filename_template** (str) – Python 3 style formatting string describing frame filenames: e.g. "frame_{:06d}.jpg" for the example dataset in the class docstring.
- **filter** (Optional[Callable[[Path], bool]]) – Optional filter callable that decides whether a given example folder is to be included in the dataset or not.
- **label_set** (Optional[LabelSet]) – Optional label set for labelling examples.
- **sampler** (*FrameSampler*) – Optional sampler for drawing frames from each video.
- **transform** (Optional[Callable[[Iterator[Image]], Tensor]]) – Optional transform performed over the loaded clip.
- **frame_counter** (Optional[Callable[[Path], int]]) – Optional callable used to determine the number of frames each video contains. The callable will be passed the path to a video folder and should return a positive integer representing the number of frames. This tends to be useful if you've precomputed the number of frames in a dataset.

__getitem__(*index*)
Load an example by index

Parameters **index** (int) – index of the example within the dataset.

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

Returns Example transformed by `transform` if one was passed during instantiation, otherwise the example is converted to a tensor without any transformations applied to it. Additionally, if a label set is present, the method return a tuple: `(video_tensor, label)`

__len__()
Total number of examples in the dataset

Return type int

1.1.3 VideoFolderDataset

class torchvideo.datasets.**VideoFolderDataset**(*root_path*, *filter=None*, *label_set=None*, *sampler=FullVideoSampler()*, *transform=None*, *frame_counter=None*)
Bases: torchvideo.datasets.video_dataset.VideoDataset

Dataset stored as a folder of videos, where each video is a single example in the dataset.

The folder hierarchy should look something like this:

```
root/video1.mp4
root/video2.mp4
...
```

Parameters

- **root_path** (Union[str, Path]) – Path to dataset folder on disk. The contents of this folder should be video files.
- **filter** (Optional[Callable[[Path], bool]]) – Optional filter callable that decides whether a given example video is to be included in the dataset or not.
- **label_set** (Optional[LabelSet]) – Optional label set for labelling examples.
- **sampler** (*FrameSampler*) – Optional sampler for drawing frames from each video.
- **transform** (Optional[Callable[[Iterator[Image]], Tensor]]) – Optional transform over the list of frames.
- **frame_counter** (Optional[Callable[[Path], int]]) – Optional callable used to determine the number of frames each video contains. The callable will be passed the path to a video and should return a positive integer representing the number of frames. This tends to be useful if you've precomputed the number of frames in a dataset.

__getitem__(*index*)
Load an example by index

Parameters **index** (int) – index of the example within the dataset.

Return type Union[Any, Tuple[Any, Any]]

Returns Example transformed by `transform` if one was passed during instantiation, otherwise the example is converted to a tensor without any transformations applied to it. Additionally, if a label set is present, the method return a tuple: `(video_tensor, label)`

__len__()
Total number of examples in the dataset

1.1.4 GulpVideoDataset

class torchvideo.datasets.**GulpVideoDataset**(*root_path*, ***, *gulp_directory=None*, *filter=None*, *label_field=None*, *label_set=None*, *sampler=FullVideoSampler()*, *transform=None*)

Bases: torchvideo.datasets.video_dataset.VideoDataset

GulpIO Video dataset.

The folder hierarchy should look something like this:

```
root/data_0.gulp
root/data_1.gulp
...

root/meta_0.gulp
root/meta_1.gulp
...
```

Parameters

- **root_path** (Union[str, Path]) – Path to GulpIO dataset folder on disk. The `.gulp` and `.gmeta` files are direct children of this directory.
- **filter** (Optional[Callable[[str], bool]]) – Filter function that determines whether a video is included into the dataset. The filter is called on each video id, and should return `True` to include the video, and `False` to exclude it.
- **label_field** (Optional[str]) – Meta data field name that stores the label of an example, this is used to construct a *GulpLabelSet* that performs the example labelling. Defaults to `'label'`.
- **label_set** (Optional[LabelSet]) – Optional label set for labelling examples. This is mutually exclusive with `label_field`.
- **sampler** (*FrameSampler*) – Optional sampler for drawing frames from each video.
- **transform** (Optional[Callable[[ndarray], Tensor]]) – Optional transform over the `ndarray` with layout `THWC`. Note you'll probably want to remap the channels to `CTHW` at the end of this transform.
- **gulp_directory** (Optional[GulpDirectory]) – Optional gulp directory residing at root_path. Useful if you wish to create a custom label_set using the gulp_directory, which you can then pass in with the gulp_directory itself to avoid reading the gulp metadata twice.

__getitem__(*index*)

Load an example by index

Parameters **index** – index of the example within the dataset.

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

Returns Example transformed by `transform` if one was passed during instantiation, otherwise the example is converted to a tensor without any transformations applied to it. Additionally, if a label set is present, the method return a tuple: `(video_tensor, label)`

__len__()

Total number of examples in the dataset

1.2 Label Sets

Label sets are an abstraction over how your video data is labelled. This provides flexibility in swapping out different storage methods and labelling methods. All datasets optionally take a *LabelSet* that performs the mapping between example and label.

1.2.1 LabelSet

class torchvideo.datasets.**LabelSet**
Bases: abc.ABC

Abstract base class that all LabelSets inherit from

If you are implementing your own LabelSet, you should inherit from this class.

__getitem__(*video_name*)

Parameters **video_name** (str) – The filename or id of the video

Return type Any

Returns The corresponding label

1.2.2 DummyLabelSet

class torchvideo.datasets.**DummyLabelSet**(*label=0*)
Bases: torchvideo.datasets.label_sets.label_set.LabelSet

A dummy label set that returns the same label regardless of video

Parameters **label** (Any) – The label given to any video

__getitem__(*video_name*)

Parameters **video_name** – The filename or id of the video

Return type Any

Returns The corresponding label

1.2.3 GulpLabelSet

class torchvideo.datasets.**GulpLabelSet**(*merged_meta_dict*, *label_field='label'*)
Bases: torchvideo.datasets.label_sets.label_set.LabelSet

LabelSet for GulpIO datasets where the label is contained within the metadata of the gulp directory. Assuming you've written the label of each video to a field called `'label'` in the metadata you can create a LabelSet like: `GulpLabelSet(gulp_dir.merged_meta_dict, label_field='label')`

__getitem__(*video_name*)

Parameters **video_name** (str) – The filename or id of the video

Return type Any

Returns The corresponding label

1.2.4 CsvLabelSet

class torchvideo.datasets.**CsvLabelSet**(*df*, *col='label'*)

Bases: torchvideo.datasets.label_sets.label_set.LabelSet

LabelSet for a pandas DataFrame or Series. The index of the DataFrame/Series is assumed to be the set of video names and the values in a series the label. For a dataframe the `field` kwarg specifies which field to use as the label

Examples

```
>>> import pandas as pd
>>> df = pd.DataFrame({'video': ['video1', 'video2'],
...                    'label': [1, 2]}).set_index('video')
>>> label_set = CsvLabelSet(df, col='label')
>>> label_set['video1']
1
```

Parameters

- **df** – pandas DataFrame or Series containing video names/ids and their corresponding labels.
- **col** (Optional[str]) – The column to read the label from when df is a DataFrame.

__getitem__(*video_name*)

Parameters **video_name** (str) – The filename or id of the video

Return type Any

Returns The corresponding label

CHAPTER

TWO

TORCHVIDEO.SAMPLERS

2.1 Samplers

Different video models use different strategies in sampling frames: some use sparse sampling strategies (like TSN, TRN) whereas others like 3D CNNs use dense sampling strategies. In order to accommodate these different architectures we offer a variety of sampling strategies with the opportunity to implement your own.

Contents

2.1.1 FrameSampler

class torchvideo.samplers.**FrameSampler**

Bases: abc.ABC

Abstract base class that all frame samplers implement.

If you are creating your own sampler, you should inherit from this base class.

sample(*video_length*)

Generate frame indices to sample from a video of `video_length` frames.

Parameters **video_length** (int) – The duration in frames of the video to be sampled from

Return type Union[slice, List[int], List[slice]]

Returns Frame indices

2.1.2 ClipSampler

class torchvideo.samplers.**ClipSampler**(*clip_length*, *frame_step=1*, *test=False*)
Bases: *torchvideo.samplers.FrameSampler*

Sample clips of a fixed duration uniformly randomly from a video.

Parameters

- **clip_length** (int) – Duration of clip in frames
- **frame_step** (int) – The step size between frames, this controls FPS reduction, a step size of 2 will halve FPS, step size of 3 will reduce FPS to 1/3.
- **test** (bool) – Whether or not to sample in test mode (in test mode the central clip is sampled from the video)

sample(*video_length*)
Generate frame indices to sample from a video of video_length frames.

Parameters **video_length** (int) – The duration in frames of the video to be sampled from

Return type Union[slice, List[int], List[slice]]

Returns Frame indices

2.1.3 FullVideoSampler

class torchvideo.samplers.**FullVideoSampler**(*frame_step=1*)
Bases: *torchvideo.samplers.FrameSampler*

Sample all frames in a video.

Parameters **frame_step** – The step size between frames, this controls FPS reduction, a step size of 2 will halve FPS, step size of 3 will reduce FPS to 1/3.

sample(*video_length*)

Args: video_length: The duration in frames of the video to be sampled from.

Returns:

a slice from 0 to video_length with step size frame_step

Return type Union[slice, List[int], List[slice]]

2.1.4 TemporalSegmentSampler

class torchvideo.samplers.**TemporalSegmentSampler**(*segment_count*, *snippet_length*, ***, *sample_count=None*, *test=False*)
Bases: *torchvideo.samplers.FrameSampler*

[TSN] style sampling.

The video is equally divided into a number of segments, segment_count and from within each segment a snippet, a contiguous sequence of frames, snippet_length fr+ames long is sampled.

There are two variants of sampling. One for training and one for testing. During training the snippet location within the segment is uniformly randomly sampled. During testing snippets are sampled centrally within their segment (i.e. deterministically).

[TSN] Uses the following configurations:

Network	Train/Test	segment_count	snippet_length
RGB	Train	3	1
	Test	25	1
Flow	Train	3	5
	Test	25	5

Parameters

- **segment_count** (int) – Number of segments to split the video into, from which a snippet is sampled.
- **snippet_length** (int) – The number of frames in each snippet
- **sample_count** (Optional[int]) – Override the number of samples to be drawn from the segments, by default the sampler will sample a total of segment_count snippets from the video. In some cases it can be useful to sample fewer than this (effectively choosing sample_count snippets from segment_count).
- **test** (bool) – Whether to sample in test mode or not (see class docstring for training/testing differences)

sample(*video_length*)

Parameters **video_length** (int) – The duration in frames of the video to be sampled from

Return type Union[List[slice], List[int]]

Returns Frame indices as list of slices

segment_video(*video_length*)

Segment a video of video_length frames into self.segment_count segments.

Parameters **video_length** (int) – num

Return type Tuple[ndarray, float]

Returns (segment_start_idx, segment_length). The segment_start_idx contains the indices of the beginning of each segment in the video. segment_length is the length for all segments.

2.1.5 LambdaSampler

class torchvideo.samplers.**LambdaSampler**(*sampler*)

Bases: *torchvideo.samplers.FrameSampler*

Custom sampler constructed from a user provided function.

Parameters **sampler** (Callable[[int], Union[slice, List[slice], List[int]]]) – Function that takes an int, the video length in frames and returns a slice, list of ints, or list of slices representing indices to sample from the video. All the indices should be less than the video length - 1.

sample(*video_length*)

Generate frame indices to sample from a video of video_length frames.

Parameters **video_length** (int) – The duration in frames of the video to be sampled from

Return type Union[slice, List[int], List[slice]]

Returns Frame indices

CHAPTER

THREE

TORCHVIDEO.TRANSFORMS

This module contains video transforms similar to those found in `torchvision.transforms` specialised for image transformations. Like the transforms from `torchvision.transforms` you can chain together successive transforms using `torchvision.transforms.Compose`.

Contents

- *time_to_channel*

3.1 Target parameters

All transforms support a *target* parameter. Currently these don't do anything, but allow you to implement transforms on targets as well as frames. At some point in future it is the intention that we'll support transforms of things like masks, or allow you to plug your own target transforms into these classes.

3.2 Examples

Typically your transformation pipelines will be compose of a sequence of PIL video transforms followed by a `CollectFrames` transform and a `PILVideoToTensor`: transform.

```
import torchvideo.transforms as VT
import torchvision.transforms as IT
from torchvision.transforms import Compose

transform = Compose([
    VT.CenterCropVideo((224, 224)),  # (h, w)
    VT.CollectFrames(),
    VT.PILVideoToTensor()
])
```

Optical flow stored as flattened (u, v) pairs like $(u_0, v_1, u_1, v_1, \ldots, u_n, v_n)$ that are then stacked into the channel dimension would be dealt with like so:

```
import torchvideo.transforms as VT
import torchvision.transforms as IT
from torchvision.transforms import Compose

transform = Compose([
    VT.CenterCropVideo((224, 224)),  # (h, w)
    VT.CollectFrames(),
    VT.PILVideoToTensor(),
    VT.TimeToChannel()
])
```

3.3 Video Datatypes

torchvideo represents videos in a variety of formats:

- *PIL video*: A list of a PIL Images, this is useful for applying image data augmentations
- *tensor video*: A `torch.Tensor` of shape (C, T, H, W) for feeding a network.
- *NDArray video*: A `numpy.ndarray` of shape either (T, H, W, C) or (C, T, H, W). The reason for the multiple channel shapes is that most loaders load in (T, H, W, C) format, however tensors formatted for input into a network typically are formatted in (C, T, H, W). Permuting the dimensions is a costly operation, so supporting both format allows for efficient implementation of transforms without have to invert the conversion from one format to the other.

3.4 Composing Transforms

Transforms can be composed with *Compose*. This functions in exactly the same way as torchvision's implementation, however it also supports chaining transforms that require, or optionally support, or don't support a target parameter. It handles the marshalling of targets around and into those transforms depending upon their support allowing you to mix transforms defined in this library (all of which support a target parameter) and those defined in other libraries.

Additionally, we provide a *IdentityTransform* that has a nicer `__repr__` suitable for use as a default transform in *Compose* pipelines.

3.4.1 Compose

class torchvideo.transforms.**Compose**(*transforms*)

Bases: object

Similar to `torchvision.transforms.transforms.Compose` except supporting transforms that take either a mandatory or optional target parameter in __call__. This facilitates chaining a mix of transforms: those that don't support target parameters, those that do, and those that require them.

__call__(*frames*, *target=<class 'torchvideo.transforms.transforms.transform.empty_target'>*)

Call self as a function.

3.4.2 IdentityTransform

class torchvideo.transforms.**IdentityTransform**

Bases: `torchvideo.transforms.transforms.transform.StatelessTransform`

Identity transformation that returns frames (and labels) unchanged. This is primarily of use when conditionally adding in transforms and you want to default to a transform that doesn't do anything. Whilst you could just use an identity lambda this transform has a nicer repr that shows that no transform is taking place.

__call__(*frames*, *target=<class 'torchvideo.transforms.transforms.transform.empty_target'>*)

Call self as a function.

3.5 Transforms on PIL Videos

These transforms all take an iterator/iterable of `PIL.Image.Image` and produce an iterator of `PIL.Image.Image`. To materialize the iterator the you should compose your sequence of PIL video transforms with *CollectFrames*.

3.5.1 CenterCropVideo

class torchvideo.transforms.**CenterCropVideo**(*size*)

Bases: `torchvideo.transforms.transforms.transform.Transform`

Crops the given video (composed of PIL Images) at the center of the frame.

Parameters **size** (*sequence or int*) – Desired output size of the crop. If size is an `int` instead of sequence like `(h, w)`, a square crop `(size, size)` is made.

__call__(*frames*, *target=<class 'torchvideo.transforms.transforms.transform.empty_target'>*)

Call self as a function.

3.5.2 RandomCropVideo

class torchvideo.transforms.**RandomCropVideo**(*size*, *padding=None*, *pad_if_needed=False*, *fill=0*, *padding_mode='constant'*)

Bases: torchvideo.transforms.transforms.transform.Transform

Crop the given Video (composed of PIL Images) at a random location.

Parameters

- **size** (Union[Tuple[int, int], int]) – Desired output size of the crop. If size is an int instead of sequence like (h, w), a square crop (size, size) is made.
- **padding** (Union[Tuple[int, int, int, int], Tuple[int, int], None]) – Optional padding on each border of the image. Default is None, i.e no padding. If a sequence of length 4 is provided, it is used to pad left, top, right, bottom borders respectively. If a sequence of length 2 is provided, it is used to pad left/right, top/bottom borders, respectively.
- **pad_if_needed** (bool) – Whether to pad the image if smaller than the desired size to avoid raising an exception.
- **fill** (int) – Pixel fill value for constant fill. If a tuple of length 3, it is used to fill R, G, B channels respectively. This value is only used when the padding_mode is 'constant'.
- **padding_mode** (str) – Type of padding. Should be one of: 'constant', 'edge', 'reflect' or 'symmetric'.
  - 'constant': pads with a constant value, this value is specified with fill.
  - 'edge': pads with the last value on the edge of the image.
  - 'reflect': pads with reflection of image (without repeating the last value on the edge) padding [1, 2, 3, 4] with 2 elements on both sides in reflect mode will result in [3, 2, 1, 2, 3, 4, 3, 2].
  - 'symmetric': pads with reflection of image (repeating the last value on the edge) padding [1, 2, 3, 4] with 2 elements on both sides in symmetric mode will result in [2, 1, 1, 2, 3, 4, 4, 3].

__call__(*frames*, *target=<class 'torchvideo.transforms.transforms.transform.empty_target'>*)

Call self as a function.

3.5.3 RandomHorizontalFlipVideo

class torchvideo.transforms.**RandomHorizontalFlipVideo**(*p=0.5*)

Bases: torchvideo.transforms.transforms.transform.Transform

Horizontally flip the given video (composed of PIL Images) randomly with a given probability p.

Parameters **p** (*float*) – probability of the image being flipped.

__call__(*frames*, *target=<class 'torchvideo.transforms.transforms.transform.empty_target'>*)

Call self as a function.

3.5.4 ResizeVideo

class torchvideo.transforms.**ResizeVideo**(*size*, *interpolation=2*)

Bases: torchvideo.transforms.transforms.transform.StatelessTransform

Resize the input video (composed of PIL Images) to the given size.

Parameters

- **size** (*sequence or int*) – Desired output size. If size is a sequence like (h, w), output size will be matched to this. If size is an int, smaller edge of the image will be matched to this number. i.e, if height > width, then image will be rescaled to (size * height / width, size).
- **interpolation** (*int, optional*) – Desired interpolation. Default is PIL.Image.BILINEAR (see PIL.Image.Image.resize() for other options).

__call__(*frames*, *target=<class 'torchvideo.transforms.transforms.transform.empty_target'>*)

Call self as a function.

3.5.5 MultiScaleCropVideo

class torchvideo.transforms.**MultiScaleCropVideo**(*size*, *scales=(1, 0.875, 0.75, 0.66)*, *max_distortion=1*, *fixed_crops=True*, *more_fixed_crops=True*)

Bases: torchvideo.transforms.transforms.transform.Transform

Random crop the input video (composed of PIL Images) at one of the given scales or from a set of fixed crops, then resize to specified size.

Parameters

- **size** (*sequence or int*) – Desired output size. If size is an int instead of sequence like (h, w), a square image (size, size) is made.
- **scales** (*sequence*) – A sequence of floats between in the range $[0, 1]$ indicating the scale of the crop to be made.
- **max_distortion** (*int*) – Integer between 0–len(scales) that controls aspect-ratio distortion. This parameters decides which scales will be combined together when creating crop boxes. A max distortion of 0 means that the crop width/height have to be from the same scale, whereas a distortion of 1 means that the crop width/height can be from 1 scale before or ahead in the scales sequence thereby stretching or squishing the frame.
- **fixed_crops** (*bool*) – Whether to use upper right, upper left, lower right, lower left and center crop positions as the list of candidate crop positions instead of those generated from scales and max_distortion.
- **more_fixed_crops** (*bool*) – Whether to add center left, center right, upper center, lower center, upper quarter left, upper quarter right, lower quarter left, lower quarter right crop positions to the list of candidate crop positions that are randomly selected. fixed_crops must be enabled to use this setting.

__call__(*frames*, *target=<class 'torchvideo.transforms.transforms.transform.empty_target'>*)

Call self as a function.

3.5.6 RandomResizedCropVideo

class torchvideo.transforms.**RandomResizedCropVideo**(*size*, *scale=(0.08, 1.0)*, *ratio=(0.75, 1.3333333333333333)*, *interpolation=2*)

Bases: torchvideo.transforms.transforms.transform.Transform

Crop the given video (composed of PIL Images) to random size and aspect ratio.

A crop of random scale (default: $[0.08, 1.0]$) of the original size and a random scale (default: $[3/4, 4/3]$) of the original aspect ratio is made. This crop is finally resized to given size. This is popularly used to train the Inception networks.

Parameters

- **size** (Union[Tuple[int, int], int]) – Desired output size. If size is an int instead of sequence like `(h, w)`, a square image `(size, size)` is made.
- **scale** (Tuple[float, float]) – range of size of the origin size cropped.
- **ratio** (Tuple[float, float]) – range of aspect ratio of the origin aspect ratio cropped.
- **interpolation** – Default: `PIL.Image.BILINEAR` (see `PIL.Image.Image.resize()` for other options).

__call__(*frames*, *target=<class 'torchvideo.transforms.transforms.transform.empty_target'>*)

Call self as a function.

3.5.7 TimeApply

class torchvideo.transforms.**TimeApply**(*img_transform*)

Bases: torchvideo.transforms.transforms.transform.StatelessTransform

Apply a PIL Image transform across time.

See torchvision.transforms for suitable *deterministic* transforms to use with meta-transform.

> **Warning:** You should only use this with deterministic image transforms. Using a transform like `torchvision.transforms.RandomCrop` will randomly crop each individual frame at a different location producing a nonsensical video.

Parameters **img_transform** (Callable[[Image], Image]) – Image transform operating on a PIL Image.

__call__(*frames*, *target=<class 'torchvideo.transforms.transforms.transform.empty_target'>*)

Call self as a function.

3.6 Transforms on Torch.*Tensor videos

These transform are applicable to *torch.*Tensor* videos only. The input to these transforms should be a tensor of shape (C, T, H, W).

3.6.1 NormalizeVideo

class torchvideo.transforms.**NormalizeVideo**(*mean*, *std*, *channel_dim=0*, *inplace=False*)

Bases: torchvideo.transforms.transforms.transform.Transform

Normalise `torch.*Tensor` t given mean: $M = (\mu_1, \ldots, \mu_n)$ and std: $\Sigma = (\sigma_1, \ldots, \sigma_n)$: $t'_c = \frac{t_c - M_c}{\Sigma_c}$

Parameters

- **mean** (Union[Sequence[Number], Number]) – Sequence of means for each channel, or a single mean applying to all channels.
- **std** (Union[Sequence[Number], Number]) – Sequence of standard deviations for each channel, or a single standard deviation applying to all channels.
- **channel_dim** (int) – Index of channel dimension. 0 for `'CTHW'` tensors and ` for `'TCHW'` tensors.
- **inplace** (bool) – Whether or not to perform the operation in place without allocating a new tensor.

__call__(*frames*, *target=<class 'torchvideo.transforms.transforms.transform.empty_target'>*)

Call self as a function.

3.6.2 TimeToChannel

class torchvideo.transforms.**TimeToChannel**

Bases: torchvideo.transforms.transforms.transform.Transform

Combine time dimension into the channel dimension by reshaping video tensor of shape (C, T, H, W) into $(C \times T, H, W)$

__call__(*frames*, *target=<class 'torchvideo.transforms.transforms.transform.empty_target'>*)

Call self as a function.

3.7 Conversion transforms

These transforms are for converting between different video representations. Typically your transformation pipeline will operate on iterators of `PIL` images which will then be aggregated by `CollectFrames` and then coverted to a tensor via `PILVideoToTensor`.

3.7.1 CollectFrames

class torchvideo.transforms.**CollectFrames**
Bases: torchvideo.transforms.transforms.transform.Transform

Collect frames from iterator into list.

Used at the end of a sequence of PIL video transformations.

__call__(*frames*, *target=<class 'torchvideo.transforms.transforms.transform.empty_target'>*)
Call self as a function.

3.7.2 PILVideoToTensor

class torchvideo.transforms.**PILVideoToTensor**(*rescale=True*, *ordering='CTHW'*)
Bases: torchvideo.transforms.transforms.transform.Transform

Convert a list of PIL Images to a tensor (C, T, H, W) or (T, C, H, W).

Parameters

- **rescale** (bool) – Whether or not to rescale video from $[0, 255]$ to $[0, 1]$. If `False` the tensor will be in range $[0, 255]$.
- **ordering** (str) – What channel ordering to convert the tensor to. Either *'CTHW'* or *'TCHW'*

__call__(*frames*, *target=<class 'torchvideo.transforms.transforms.transform.empty_target'>*)
Call self as a function.

3.7.3 NDArrayToPILVideo

class torchvideo.transforms.**NDArrayToPILVideo**(*format='thwc'*)
Bases: torchvideo.transforms.transforms.transform.Transform

Convert numpy.ndarray of the format (T, H, W, C) or (C, T, H, W) to a PIL video (an iterator of PIL images)

Parameters **format** – dimensional layout of array, one of `"thwc"` or `"cthw"`

__call__(*frames*, *target=<class 'torchvideo.transforms.transforms.transform.empty_target'>*)
Call self as a function.

3.8 Functional Transforms

Functional transforms give you fine-grained control of the transformation pipeline. As opposed to the transformations above, functional transforms don't contain a random number generator for their parameters.

3.8.1 normalize

torchvideo.transforms.functional.**normalize**(*tensor*, *mean*, *std*, *channel_dim=0*, *inplace=False*)

Channel-wise normalize a tensor video of shape (C, T, H, W) with mean and standard deviation

See *NormalizeVideo* for more details.

Parameters

- **tensor** (Tensor) – Tensor video of size (C, T, H, W) to be normalized.
- **mean** (Sequence) – Sequence of means, M, for each channel c.
- **std** (Sequence) – Sequence of standard deviations, Σ, for each channel c.
- **channel_dim** (int) – Index of channel dimension. 0 for `'CTHW'` tensors and ` for `'TCHW'` tensors.
- **inplace** (bool) – Whether to normalise the tensor without cloning or not.

Return type Tensor

Returns Channel-wise normalised tensor video, $t'_c = \frac{t_c - M_c}{\Sigma_c}$

3.8.2 time_to_channel

torchvideo.transforms.functional.**time_to_channel**(*tensor*)

Reshape video tensor of shape (C, T, H, W) into $(C \times T, H, W)$

Parameters **tensor** (Tensor) – Tensor video of size (C, T, H, W)

Return type Tensor

Returns Tensor of shape $(C \times T, H, W)$

CHAPTER

FOUR

TORCHVIDEO.TOOLS

4.1 Tools

torchvideo.tools.**show_video**(*frames*, *fps=30*, *ndarray_format='THWC'*)

Show `frames` as a video in Jupyter, or in a PyGame window using `moviepy`.

Parameters

- **frames** (Union[Tensor, ndarray, List[Image]]) – One of:
  - `torch.Tensor` with layout `CTHW`.
  - `numpy.ndarray` of layout `THWC` or `CTHW`, if the latter, then set `ndarray_format` to `CTHW`. The array should have a `np.uint8` dtype and range `[0, 255]`.
  - a list of `PIL.Image.Image`.
- **fps** (*optional*) – Frame rate of video
- **ndarray_format** – 'CTHW' or 'THWC' depending on layout of ndarray.

Returns ImageSequenceClip displayed.

torchvideo.tools.**convert_to_clip**(*frames*, *fps=30*, *ndarray_format='THWC'*)

Convert `frames` to a `moviepy` `ImageSequenceClip`.

Parameters

- **frames** – One of:
  - `torch.Tensor` with layout `CTHW`.
  - `numpy.ndarray` of layout `THWC` or `CTHW`, if the latter, then set `ndarray_format` to `CTHW`. The array should have a `np.uint8` dtype and range `[0, 255]`.
  - a list of `PIL.Image.Image`.
- **fps** (*optional*) – Frame rate of video
- **ndarray_format** – 'CTHW' or 'THWC' depending on layout of ndarray.

Returns ImageSequenceClip

CHAPTER

FIVE

BIBLIOGRAPHY

CHAPTER

SIX

INSTALLATION

Install torchvideo from PyPI with:

```
$ pip install torchvideo
```

or the cutting edge branch from github with:

```
$ pip install git+https://github.com/willprice/torchvideo.git
```

We **strongly** advise you to install Pillow-simd to speed up image transformations. Do this after installing torchvideo.

```
$ pip install pillow-simd
```

BIBLIOGRAPHY

[TSN] Temporal Segment Networks: Towards Good Practices for Deep Action Recognition, Limin Wang, Yuanjun Xiong, Zhe Wang, Yu Qiao, Dahua Lin, Xiaoou Tang, Luc Van Gool

PYTHON MODULE INDEX

t

INDEX

Symbols

C

D

F

G

I

L