Load video data

> [!WARNING]
> Video support is experimental and is subject to change.

Video datasets have Video type columns, which contain `torchvision` objects.

> [!TIP]
> To work with video datasets, you need to have the `torchvision` and `av` packages installed. Check out the installation guide to learn how to install them.

When you load a video dataset and call the video column, the videos are decoded as `torchvision` Videos:

```
>>> from datasets import load_dataset, Video

>>> dataset = load_dataset("path/to/video/folder", split="train")
>>> dataset[0]["video"]
<torchcodec.decoders._video_decoder.VideoDecoder object at 0x14a61d5a0>
```

> [!WARNING]
> Index into a video dataset using the row index first and then the `video` column - `dataset[0]["video"]` - to avoid creating all the video objects in the dataset. Otherwise, this can be a slow and time-consuming process if you have a large dataset.

For a guide on how to load any type of dataset, take a look at the general loading guide.

Read frames

Access frames directly from a video using the `VideoReader` using `next()`:

```
>>> video = dataset[0]["video"]
>>> first_frame = video.get_frame_at(0)
>>> first_frame.data.shape
(3, 240, 320)
>>> first_frame.pts_seconds  # timestamp
0.0
```

To get multiple frames at once, you can call `.get_frames_in_range(start: int, stop: int, step: int)`. This will return a frame batch. This is the efficient way to obtain a long list of frames refer to the torchcodec docs to see more functions for effiently accessing the data:

```
>>> import torch
>>> frames = video.get_frames_in_range(0, 6, 1)
>>> frames.data.shape
torch.Size([5, 3, 240, 320])
```

There is also `.get_frames_played_in_range(start_seconds: float, stop_seconds: float)` to access all frames played whithin a certain time range.

```
>>> frames = video.get_frames_played_in_range(.5, 1.2)
>>> frames.data.shape
torch.Size([42, 3, 240, 320])
```

Local files

You can load a dataset from the video path. Use the cast_column() function to accept a column of video file paths, and decode it into a `torchcodec` video with the Video feature:

```
>>> from datasets import Dataset, Video

>>> dataset = Dataset.from_dict({"video": ["path/to/video_1", "path/to/video_2", ..., "path/to/vid
>>> dataset[0]["video"]
<torchcodec.decoders._video_decoder.VideoDecoder object at 0x14a61e080>
```

If you only want to load the underlying path to the video dataset without decoding the video object, set `decode=False` in the Video feature:

```
>>> dataset = dataset.cast_column("video", Video(decode=False))
>>> dataset[0]["video"]
{'bytes': None,
 'path': 'path/to/video/folder/video0.mp4'}
```

VideoFolder

You can also load a dataset with an `VideoFolder` dataset builder which does not require writing a custom dataloader. This makes `VideoFolder` ideal for quickly creating and loading video datasets with several thousand videos for different vision tasks. Your video dataset structure should look like this:

```
folder/train/dog/golden_retriever.mp4
folder/train/dog/german_shepherd.mp4
folder/train/dog/chihuahua.mp4

folder/train/cat/maine_coon.mp4
folder/train/cat/bengal.mp4
folder/train/cat/birman.mp4
```

If the dataset follows the `VideoFolder` structure, then you can load it directly with load_dataset():

```
>>> from datasets import load_dataset

>>> dataset = load_dataset("username/dataset_name")
>>> # OR locally:
>>> dataset = load_dataset("/path/to/folder")
```

For local datasets, this is equivalent to passing `videofolder` manually in load_dataset() and the directory in `data_dir`:

```
>>> dataset = load_dataset("videofolder", data_dir="/path/to/folder")
```

Then you can access the videos as `torchcodec.decoders._video_decoder.VideoDecoder` objects:

```
>>> dataset["train"][0]
{"video": <torchcodec.decoders._video_decoder.VideoDecoder object at 0x14a61e080>, "label": 0}

>>> dataset["train"][-1]
{"video": <torchcodec.decoders._video_decoder.VideoDecoder object at 0x14a61e090>, "label": 1}
```

To ignore the information in the metadata file, set `drop_metadata=True` in load_dataset():

```
>>> from datasets import load_dataset

>>> dataset = load_dataset("username/dataset_with_metadata", drop_metadata=True)
```

If you don't have a metadata file, `VideoFolder` automatically infers the label name from the directory name.
If you want to drop automatically created labels, set `drop_labels=True`.
In this case, your dataset will only contain a video column:

```
>>> from datasets import load_dataset

>>> dataset = load_dataset("username/dataset_without_metadata", drop_labels=True)
```

Finally the `filters` argument lets you load only a subset of the dataset, based on a condition on the label or the metadata. This is especially useful if the metadata is in Parquet format, since this format enables fast filtering. It is also recommended to use this argument with `streaming=True`, because by default the dataset is fully downloaded before filtering.

```
>>> filters = [("label", "=", 0)]
>>> dataset = load_dataset("username/dataset_name", streaming=True, filters=filters)
```

> [!TIP]

> For more information about creating your own `VideoFolder` dataset, take a look at the Create a video dataset guide.

WebDataset

The WebDataset format is based on a folder of TAR archives and is suitable for big video datasets.
Because of their size, WebDatasets are generally loaded in streaming mode (using `streaming=True`).

You can load a WebDataset like this:

```
>>> from datasets import load_dataset

>>> dataset = load_dataset("webdataset", data_dir="/path/to/folder", streaming=True)
```

Video decoding

By default, videos are decoded sequentially as torchvision `VideoReaders` when you iterate on a dataset.
It sequentially decodes the metadata of the videos, and doesn't read the video frames until you access them.

However it is possible to speed up the dataset significantly using multithreaded decoding:

```
>>> import os
>>> num_threads = num_threads = min(32, (os.cpu_count() or 1) + 4)
>>> dataset = dataset.decode(num_threads=num_threads)
>>> for example in dataset:  # up to 20 times faster !
...     ...
```

You can enable multithreading using `num_threads`. This is especially useful to speed up remote data streaming.
However it can be slower than `num_threads=0` for local data on fast disks.

If you are not interested in the videos decoded as torchvision `VideoReaders` and would like to

access the path/bytes instead, you can disable decoding:

```
>>> dataset = dataset.decode(False)
```

Note: IterableDataset.decode() is only available for streaming datasets at the moment.