Use with PyTorch

This document is a quick introduction to using `datasets` with PyTorch, with a particular focus on how to get
`torch.Tensor` objects out of our datasets, and how to use a PyTorch `DataLoader` and a Hugging Face `Dataset`
with the best performance.

Dataset format

By default, datasets return regular python objects: integers, floats, strings, lists, etc.

To get PyTorch tensors instead, you can set the format of the dataset to `pytorch` using Dataset.with_format():

```
>>> from datasets import Dataset
>>> data = [[1, 2],[3, 4]]
>>> ds = Dataset.from_dict({"data": data})
>>> ds = ds.with_format("torch")
>>> ds[0]
{'data': tensor([1, 2])}
>>> ds[:2]
{'data': tensor([[1, 2],
         [3, 4]])}
```

> [!TIP]
> A Dataset object is a wrapper of an Arrow table, which allows fast zero-copy reads from arrays in the dataset to PyTorch tensors.

To load the data as tensors on a GPU, specify the `device` argument:

```
>>> import torch
>>> device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
>>> ds = ds.with_format("torch", device=device)
>>> ds[0]
{'data': tensor([1, 2], device='cuda:0')}
```

N-dimensional arrays

If your dataset consists of N-dimensional arrays, you will see that by default they are considered as the same tensor if the shape is fixed:

```
>>> from datasets import Dataset
>>> data = [[[1, 2],[3, 4]],[[5, 6],[7, 8]]]  # fixed shape
>>> ds = Dataset.from_dict({"data": data})
>>> ds = ds.with_format("torch")
>>> ds[0]
{'data': tensor([[1, 2],
         [3, 4]])}
```

```
>>> from datasets import Dataset
>>> data = [[[1, 2],[3]],[[4, 5, 6],[7, 8]]]  # varying shape
>>> ds = Dataset.from_dict({"data": data})
>>> ds = ds.with_format("torch")
>>> ds[0]
{'data': [tensor([1, 2]), tensor([3])]}
```

However this logic often requires slow shape comparisons and data copies.
To avoid this, you must explicitly use the `Array` feature type and specify the shape of your tensors:

```
>>> from datasets import Dataset, Features, Array2D
>>> data = [[[1, 2],[3, 4]],[[5, 6],[7, 8]]]
>>> features = Features({"data": Array2D(shape=(2, 2), dtype='int32')})
>>> ds = Dataset.from_dict({"data": data}, features=features)
>>> ds = ds.with_format("torch")
>>> ds[0]
{'data': tensor([[1, 2],
         [3, 4]])}
>>> ds[:2]
{'data': tensor([[[1, 2],
          [3, 4]],

         [[5, 6],
          [7, 8]]])}
```

Other feature types

ClassLabel data are properly converted to tensors:

```
>>> from datasets import Dataset, Features, ClassLabel
>>> labels = [0, 0, 1]
>>> features = Features({"label": ClassLabel(names=["negative", "positive"])})
>>> ds = Dataset.from_dict({"label": labels}, features=features)
>>> ds = ds.with_format("torch")
>>> ds[:3]
{'label': tensor([0, 0, 1])}
```

String and binary objects are unchanged, since PyTorch only supports numbers.

The Image and Audio feature types are also supported.

> [!TIP]
> To use the Image feature type, you'll need to install the `vision` extra as `pip install datasets[vision]` .

```
>>> from datasets import Dataset, Features, Audio, Image
>>> images = ["path/to/image.png"] * 10
>>> features = Features({"image": Image()})
>>> ds = Dataset.from_dict({"image": images}, features=features)
>>> ds = ds.with_format("torch")
>>> ds[0]["image"].shape
torch.Size([512, 512, 4])
>>> ds[0]
{'image': tensor([[[255, 215, 106, 255],
         [255, 215, 106, 255],
         ...,
         [255, 255, 255, 255],
         [255, 255, 255, 255]]], dtype=torch.uint8)}
>>> ds[:2]["image"].shape
torch.Size([2, 512, 512, 4])
>>> ds[:2]
{'image': tensor([[[[255, 215, 106, 255],
          [255, 215, 106, 255],
          ...,
          [255, 255, 255, 255],
          [255, 255, 255, 255]]]], dtype=torch.uint8)}
```

> [!TIP]
> To use the Audio feature type, you'll need to install the `audio` extra as `pip install datasets[audio]`.

```
>>> from datasets import Dataset, Features, Audio, Image
>>> audio = ["path/to/audio.wav"] * 10
>>> features = Features({"audio": Audio()})
>>> ds = Dataset.from_dict({"audio": audio}, features=features)
>>> ds = ds.with_format("torch")
>>> ds[0]["audio"]["array"]
tensor([ 6.1035e-05,  1.5259e-05,  1.6785e-04,  ..., -1.5259e-05,
        -1.5259e-05,  1.5259e-05])
>>> ds[0]["audio"]["sampling_rate"]
tensor(44100)
```

Data loading

Like `torch.utils.data.Dataset` objects, a Dataset can be passed directly to a PyTorch `DataLoader`:

```
>>> import numpy as np
>>> from datasets import Dataset
>>> from torch.utils.data import DataLoader
>>> data = np.random.rand(16)
>>> label = np.random.randint(0, 2, size=16)
>>> ds = Dataset.from_dict({"data": data, "label": label}).with_format("torch")
>>> dataloader = DataLoader(ds, batch_size=4)
>>> for batch in dataloader:
...     print(batch)
{'data': tensor([0.0047, 0.4979, 0.6726, 0.8105]), 'label': tensor([0, 1, 0, 1])}
{'data': tensor([0.4832, 0.2723, 0.4259, 0.2224]), 'label': tensor([0, 0, 0, 0])}
{'data': tensor([0.5837, 0.3444, 0.4658, 0.6417]), 'label': tensor([0, 1, 0, 0])}
{'data': tensor([0.7022, 0.1225, 0.7228, 0.8259]), 'label': tensor([1, 1, 1, 1])}
```

Optimize data loading

There are several ways you can increase the speed your data is loaded which can save you time, especially if you are working with large datasets.
PyTorch offers parallelized data loading, retrieving batches of indices instead of individually, and streaming to iterate over the dataset without downloading it on disk.

Use multiple Workers

You can parallelize data loading with the `num_workers` argument of a PyTorch `DataLoader` and get a higher throughput.

Under the hood, the `DataLoader` starts `num_workers` processes.
Each process reloads the dataset passed to the `DataLoader` and is used to query examples.
Reloading the dataset inside a worker doesn't fill up your RAM, since it simply memory-maps the dataset again from your disk.

```
>>> import numpy as np
>>> from datasets import Dataset, load_from_disk
>>> from torch.utils.data import DataLoader
>>> data = np.random.rand(10_000)
>>> Dataset.from_dict({"data": data}).save_to_disk("my_dataset")
>>> ds = load_from_disk("my_dataset").with_format("torch")
>>> dataloader = DataLoader(ds, batch_size=32, num_workers=4)
```

Stream data

Stream a dataset by loading it as an IterableDataset. This allows you to progressively iterate over a remote dataset without downloading it on disk and or over local data files.
Learn more about which type of dataset is best for your use case in the choosing between a regular dataset or an iterable dataset guide.

An iterable dataset from `datasets` inherits from `torch.utils.data.IterableDataset` so you can pass it to a `torch.utils.data.DataLoader`:

```
>>> import numpy as np
>>> from datasets import Dataset, load_dataset
>>> from torch.utils.data import DataLoader
>>> data = np.random.rand(10_000)
>>> Dataset.from_dict({"data": data}).push_to_hub("<username>/my_dataset")  # Upload to the Huggin
>>> my_iterable_dataset = load_dataset("<username>/my_dataset", streaming=True, split="train")
>>> dataloader = DataLoader(my_iterable_dataset, batch_size=32)
```

If the dataset is split in several shards (i.e. if the dataset consists of multiple data files), then you can stream in parallel using `num_workers`:

```
>>> my_iterable_dataset = load_dataset("deepmind/code_contests", streaming=True, split="train")
>>> my_iterable_dataset.num_shards
39
>>> dataloader = DataLoader(my_iterable_dataset, batch_size=32, num_workers=4)
```

In this case each worker is given a subset of the list of shards to stream from.

Checkpoint and resume

If you need a DataLoader that you can checkpoint and resume in the middle of training, you can use the `StatefulDataLoader` from torchdata:

```
>>> from torchdata.stateful_dataloader import StatefulDataLoader
>>> my_iterable_dataset = load_dataset("deepmind/code_contests", streaming=True, split="train")
>>> dataloader = StatefulDataLoader(my_iterable_dataset, batch_size=32, num_workers=4)
>>> # save in the middle of training
>>> state_dict = dataloader.state_dict()
>>> # and resume later
>>> dataloader.load_state_dict(state_dict)
```

This is possible thanks to IterableDataset.state_dict() and IterableDataset.load_state_dict().

Distributed

To split your dataset across your training nodes, you can use datasets.distributed.split_dataset_by_node():

```
import os
from datasets.distributed import split_dataset_by_node

ds = split_dataset_by_node(ds, rank=int(os.environ["RANK"]), world_size=int(os.environ["WORLD_SIZE
```

This works for both map-style datasets and iterable datasets.
The dataset is split for the node at rank `rank` in a pool of nodes of size `world_size`.

For map-style datasets:

Each node is assigned a chunk of data, e.g. rank 0 is given the first chunk of the dataset.

For iterable datasets:

If the dataset has a number of shards that is a factor of `world_size` (i.e. if `dataset.num_shards % world_size == 0`),
then the shards are evenly assigned across the nodes, which is the most optimized.
Otherwise, each node keeps 1 example out of `world_size`, skipping the other examples.

This can also be combined with a `torch.utils.data.DataLoader` if you want each node to use multiple workers to load the data.