Differences between Dataset and IterableDataset

There are two types of dataset objects, a Dataset and an IterableDataset.
Whichever type of dataset you choose to use or create depends on the size of the dataset.
In general, an IterableDataset is ideal for big datasets (think hundreds of GBs!) due to its lazy behavior and speed advantages, while a Dataset is great for everything else.
This page will compare the differences between a Dataset and an IterableDataset to help you pick the right dataset object for you.

Downloading and streaming

When you have a regular Dataset, you can access it using `my_dataset[0]`. This provides random access to the rows.
Such datasets are also called "map-style" datasets.
For example you can download ImageNet-1k like this and access any row:

```
from datasets import load_dataset

imagenet = load_dataset("timm/imagenet-1k-wds", split="train")  # downloads the full dataset
print(imagenet[0])
```

But one caveat is that you must have the entire dataset stored on your disk or in memory, which blocks you from accessing datasets bigger than the disk.
Because it can become inconvenient for big datasets, there exists another type of dataset, the IterableDataset.
When you have an `IterableDataset`, you can access it using a `for` loop to load the data progressively as you iterate over the dataset.
This way, only a small fraction of examples is loaded in memory, and you don't write anything on disk.

For example, you can stream the ImageNet-1k dataset without downloading it on disk:

```python
from datasets import load_dataset

imagenet = load_dataset("timm/imagenet-1k-wds", split="train", streaming=True)  # will start loadi
for example in imagenet:
    print(example)
    break
```

Streaming can read online data without writing any file to disk.
For example, you can stream datasets made out of multiple shards, each of which is hundreds of gigabytes like C4 or LAION-2B.
Learn more about how to stream a dataset in the Dataset Streaming Guide.

This is not the only difference though, because the "lazy" behavior of an `IterableDataset` is also present when it comes to dataset creation and processing.

Creating map-style datasets and iterable datasets

You can create a Dataset using lists or dictionaries, and the data is entirely converted to Arrow so you can easily access any row:

```python
my_dataset = Dataset.from_dict({"col_1": [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]})
print(my_dataset[0])
```

To create an `IterableDataset` on the other hand, you must provide a "lazy" way to load the data.
In Python, we generally use generator functions. These functions `yield` one example at a time, which means you can't access a row by slicing it like a regular `Dataset`:

```python
def my_generator(n):
    for i in range(n):
        yield {"col_1": i}

my_iterable_dataset = IterableDataset.from_generator(my_generator, gen_kwargs={"n": 10})
for example in my_iterable_dataset:
    print(example)
    break
```

Loading local files entirely and progressively

It is possible to convert local or remote data files to an Arrow Dataset using load_dataset():

```
data_files = {"train": ["path/to/data.csv"]}
my_dataset = load_dataset("csv", data_files=data_files, split="train")
print(my_dataset[0])
```

However, this requires a conversion step from CSV to Arrow format, which takes time and disk space if your dataset is big.

To save disk space and skip the conversion step, you can define an `IterableDataset` by streaming from the local files directly.
This way, the data is read progressively from the local files as you iterate over the dataset:

```
data_files = {"train": ["path/to/data.csv"]}
my_iterable_dataset = load_dataset("csv", data_files=data_files, split="train", streaming=True)
for example in my_iterable_dataset:  # this reads the CSV file progressively as you iterate over t
    print(example)
    break
```

Many file formats are supported, like CSV, JSONL, and Parquet, as well as image and audio files.
You can find more information in the corresponding guides for loading tabular, text, vision, and audio datasets.

Eager data processing and lazy data processing

When you process a Dataset object using Dataset.map(), the entire dataset is processed immediately and returned.
This is similar to how `pandas` works for example.

```
my_dataset = my_dataset.map(process_fn)  # process_fn is applied on all the examples of the datase
print(my_dataset[0])
```

On the other hand, due to the "lazy" nature of an `IterableDataset`, calling IterableDataset.map() does not apply your `map` function over the full dataset.
Instead, your `map` function is applied on-the-fly.

Because of that, you can chain multiple processing steps and they will all run at once when you start iterating over the dataset:

```
my_iterable_dataset = my_iterable_dataset.map(process_fn_1)
my_iterable_dataset = my_iterable_dataset.filter(filter_fn)
my_iterable_dataset = my_iterable_dataset.map(process_fn_2)

# process_fn_1, filter_fn and process_fn_2 are applied on-the-fly when iterating over the dataset
for example in my_iterable_dataset:
    print(example)
    break
```

Exact and fast approximate shuffling

When you shuffle a Dataset using Dataset.shuffle(), you apply an exact shuffling of the dataset.
It works by taking a list of indices `[0, 1, 2, ... len(my_dataset) - 1]` and shuffling this list.
Then, accessing `my_dataset[0]` returns the row and index defined by the first element of the indices mapping that has been shuffled:

```
my_dataset = my_dataset.shuffle(seed=42)
print(my_dataset[0])
```

Since we don't have random access to the rows in the case of an `IterableDataset`, we can't use a shuffled list of indices and access a row at an arbitrary position.
This prevents the use of exact shuffling.
Instead, a fast approximate shuffling is used in IterableDataset.shuffle().
It uses a shuffle buffer to sample random examples iteratively from the dataset.
Since the dataset is still read iteratively, it provides excellent speed performance:

```python
my_iterable_dataset = my_iterable_dataset.shuffle(seed=42, buffer_size=100)
for example in my_iterable_dataset:
    print(example)
    break
```

But using a shuffle buffer is not enough to provide a satisfactory shuffling for machine learning model training. So IterableDataset.shuffle() also shuffles the dataset shards if your dataset is made of multiple files or sources:

```python
# Stream from the internet
my_iterable_dataset = load_dataset("deepmind/code_contests", split="train", streaming=True)
my_iterable_dataset.num_shards  # 39

# Stream from local files
data_files = {"train": [f"path/to/data_{i}.csv" for i in range(1024)]}
my_iterable_dataset = load_dataset("csv", data_files=data_files, split="train", streaming=True)
my_iterable_dataset.num_shards  # 1024

# From a generator function
def my_generator(n, sources):
    for source in sources:
        for example_id_for_current_source in range(n):
            yield {"example_id": f"{source}_{example_id_for_current_source}"}

gen_kwargs = {"n": 10, "sources": [f"path/to/data_{i}" for i in range(1024)]}
my_iterable_dataset = IterableDataset.from_generator(my_generator, gen_kwargs=gen_kwargs)
my_iterable_dataset.num_shards  # 1024
```

Speed differences

Regular Dataset objects are based on Arrow which provides fast random access to the rows. Thanks to memory mapping and the fact that Arrow is an in-memory format, reading data from disk doesn't do expensive system calls and deserialization.
It provides even faster data loading when iterating using a `for` loop by iterating on contiguous Arrow record batches.

However as soon as your Dataset has an indices mapping (via Dataset.shuffle() for example),

the speed can become 10x slower.
This is because there is an extra step to get the row index to read using the indices mapping, and most importantly, you aren't reading contiguous chunks of data anymore.
To restore the speed, you'd need to rewrite the entire dataset on your disk again using Dataset.flatten_indices(), which removes the indices mapping.
This may take a lot of time depending on the size of your dataset though:

```
my_dataset[0]  # fast
my_dataset = my_dataset.shuffle(seed=42)
my_dataset[0]  # up to 10x slower
my_dataset = my_dataset.flatten_indices()  # rewrite the shuffled dataset on disk as contiguous ch
my_dataset[0]  # fast again
```

In this case, we recommend switching to an IterableDataset and leveraging its fast approximate shuffling method IterableDataset.shuffle().
It only shuffles the shards order and adds a shuffle buffer to your dataset, which keeps the speed of your dataset optimal.
You can also reshuffle the dataset easily:

```
for example in enumerate(my_iterable_dataset):  # fast
    pass

shuffled_iterable_dataset = my_iterable_dataset.shuffle(seed=42, buffer_size=100)

for example in enumerate(shuffled_iterable_dataset):  # as fast as before
    pass

shuffled_iterable_dataset = my_iterable_dataset.shuffle(seed=1337, buffer_size=100)  # reshuffling

for example in enumerate(shuffled_iterable_dataset):  # still as fast as before
    pass
```

If you're using your dataset on multiple epochs, the effective seed to shuffle the shards order in the shuffle buffer is `seed + epoch`.
It makes it easy to reshuffle a dataset between epochs:

```python
for epoch in range(n_epochs):
    my_iterable_dataset.set_epoch(epoch)
    for example in my_iterable_dataset:  # fast + reshuffled at each epoch using `effective_seed =
        pass
```

To restart the iteration of a map-style dataset, you can simply skip the first examples:

```python
my_dataset = my_dataset.select(range(start_index, len(dataset)))
```

But if you use a `DataLoader` with a `Sampler`, you should instead save the state of your sampler (you might have written a custom sampler that allows resuming).

On the other hand, iterable datasets don't provide random access to a specific example index to resume from. But you can use IterableDataset.state_dict() and IterableDataset.load_state_dict() to resume from a checkpoint instead, similarly to what you can do for models and optimizers:

```python
>>> iterable_dataset = Dataset.from_dict({"a": range(6)}).to_iterable_dataset(num_shards=3)
>>> # save in the middle of training
>>> state_dict = iterable_dataset.state_dict()
>>> # and resume later
>>> iterable_dataset.load_state_dict(state_dict)
```

Under the hood, the iterable dataset keeps track of the current shard being read and the example index in the current shard and it stores this info in the `state_dict`.

To resume from a checkpoint, the dataset skips all the shards that were previously read to restart from the current shard.
Then it reads the shard and skips examples until it reaches the exact example from the checkpoint.

Therefore restarting a dataset is quite fast, since it will not re-read the shards that have already been iterated on. Still, resuming a dataset is generally not instantaneous since it has to restart reading from the beginning of the current shard and skip examples until it reaches the checkpoint location.

This can be used with the `StatefulDataLoader` from `torchdata`, see streaming with a PyTorch

DataLoader.

Switch from map-style to iterable

If you want to benefit from the "lazy" behavior of an IterableDataset or their speed advantages, you can switch your map-style Dataset to an IterableDataset:

```
my_iterable_dataset = my_dataset.to_iterable_dataset()
```

If you want to shuffle your dataset or use it with a PyTorch DataLoader, we recommend generating a sharded IterableDataset:

```
my_iterable_dataset = my_dataset.to_iterable_dataset(num_shards=1024)
my_iterable_dataset.num_shards  # 1024
```