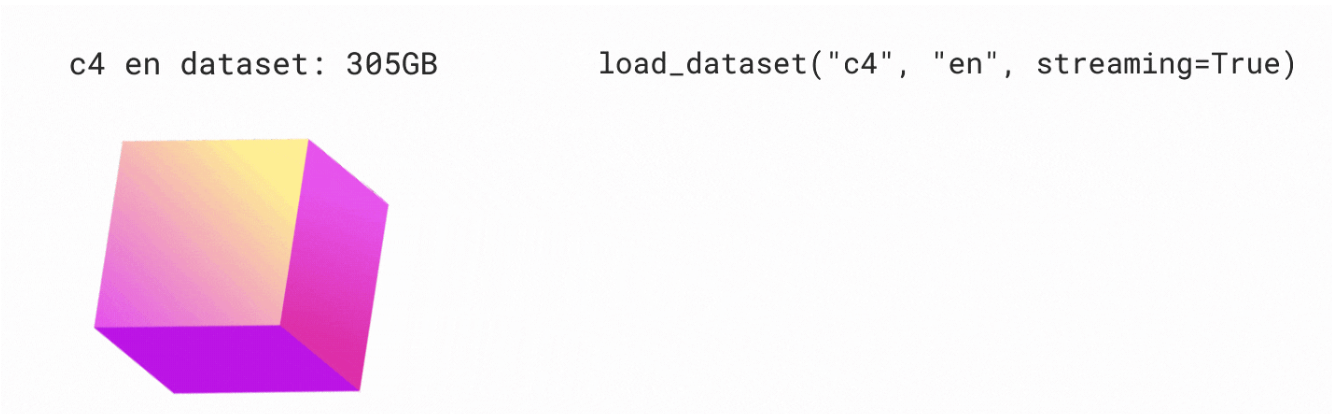
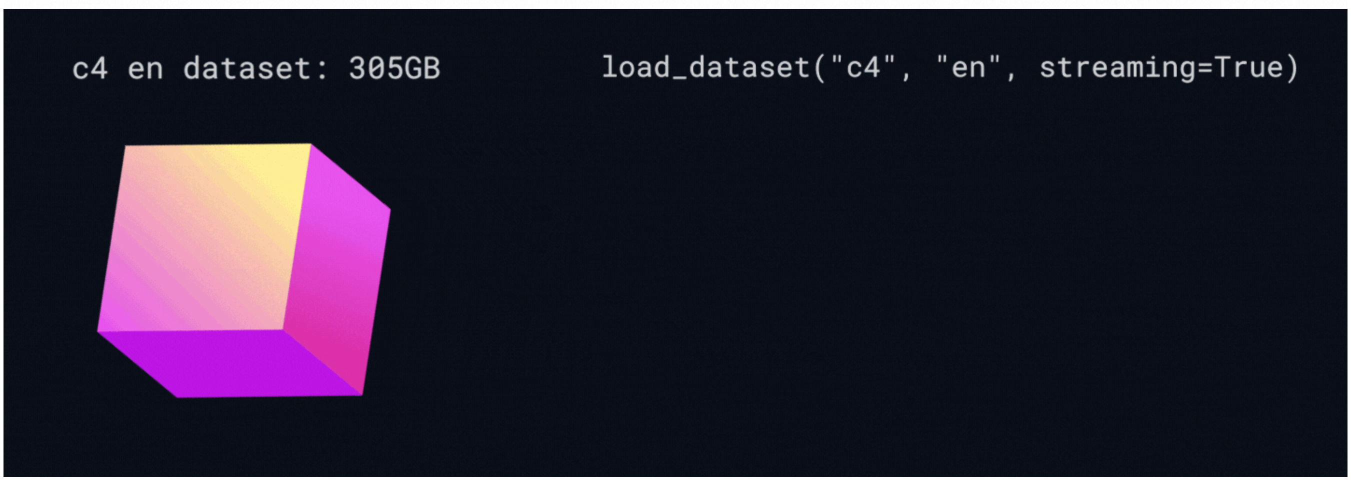
Stream

Dataset streaming lets you work with a dataset without downloading it.
The data is streamed as you iterate over the dataset.
This is especially helpful when:

- You don't want to wait for an extremely large dataset to download.
- The dataset size exceeds the amount of available disk space on your computer.
- You want to quickly explore just a few samples of a dataset.





For example, the English split of the HuggingFaceFW/fineweb dataset is 45 terabytes, but you can use it instantly with streaming. Stream a dataset by setting `streaming=True` in load_dataset() as shown below:

```
>>> from datasets import load_dataset
>>> dataset = load_dataset('HuggingFaceFW/fineweb', split='train', streaming=True)
>>> print(next(iter(dataset)))
{'text': 'How AP reported in all formats from tornado-stricken regionsMarch 8, 2012\nWhen the firs
 'language_score': 0.9721424579620361, 'token_count': 717}
```

Dataset streaming also lets you work with a dataset made of local files without doing any conversion.
In this case, the data is streamed from the local files as you iterate over the dataset.
This is especially helpful when:

- You don't want to wait for an extremely large local dataset to be converted to Arrow.
- The converted files size would exceed the amount of available disk space on your computer.
- You want to quickly explore just a few samples of a dataset.
- You want to load only certain columns or efficiently filter a Parquet dataset.

For example, you can stream a local dataset of hundreds of compressed JSONL files like oscar-corpus/OSCAR-2201 to use it instantly:

```
>>> from datasets import load_dataset
>>> data_files = {'train': 'path/to/OSCAR-2201/compressed/en_meta/*.jsonl.gz'}
>>> dataset = load_dataset('json', data_files=data_files, split='train', streaming=True)
>>> print(next(iter(dataset)))
{'id': 0, 'text': 'Founded in 2015, Golden Bees is a leading programmatic recruitment platform ded
```

Parquet is a columnar format that allows you to stream and load only a subset of columns and ignore unwanted columns. Parquet also stores metadata such as column statistics (at the file and row group level), enabling efficient filtering. Use the `columns` and `filters` arguments of datasets.packaged_modules.parquet.ParquetConfig to stream Parquet datasets, select columns, and apply filters:

```
>>> from datasets import load_dataset
>>> dataset = load_dataset('HuggingFaceFW/fineweb', split='train', streaming=True, columns=["url",
>>> print(next(iter(dataset)))
{'url': 'http://%20jwashington@ap.org/Content/Press-Release/2012/How-AP-reported-in-all-formats-fr
>>> dataset = load_dataset('HuggingFaceFW/fineweb', split='train', streaming=True, filters=[("lang
>>> print(next(iter(dataset)))
{'text': 'Everyone wishes for something. And lots of people believe they know how to make their wi
 'language_score': 0.9900368452072144, 'token_count': 716}
```

Loading a dataset in streaming mode creates a new dataset type instance (instead of the classic Dataset object), known as an IterableDataset.
This special type of dataset has its own set of processing methods shown below.

> [!TIP]
> An IterableDataset is useful for iterative jobs like training a model.
> You shouldn't use a IterableDataset for jobs that require random access to examples because you have to iterate all over it using a for loop. Getting the last example in an iterable dataset would require you to iterate over all the previous examples.
> You can find more details in the Dataset vs. IterableDataset guide.

Column indexing

Sometimes it is convenient to iterate over values of a specific column. Fortunately, an IterableDataset supports column indexing:

```
>>> from datasets import load_dataset
>>> dataset = load_dataset("allenai/c4", "en", streaming=True, split="train")
>>> print(next(iter(dataset["text"])))
Beginners BBQ Class Taking Place in Missoula!...
```

Convert from a Dataset

If you have an existing Dataset object, you can convert it to an IterableDataset with the to_iterable_dataset() function. This is actually faster than setting the `streaming=True` argument in load_dataset() because the data is streamed from local files.

```
>>> from datasets import load_dataset

# faster 🐇
>>> dataset = load_dataset("ethz/food101")
>>> iterable_dataset = dataset.to_iterable_dataset()

# slower 🐢
>>> iterable_dataset = load_dataset("ethz/food101", streaming=True)
```

The to_iterable_dataset() function supports sharding when the IterableDataset is instantiated. This is useful when working with big datasets, and you'd like to shuffle the dataset or to enable fast parallel loading with a PyTorch DataLoader.

```
>>> import torch
>>> from datasets import load_dataset

>>> dataset = load_dataset("ethz/food101")
>>> iterable_dataset = dataset.to_iterable_dataset(num_shards=64) # shard the dataset
>>> iterable_dataset = iterable_dataset.shuffle(buffer_size=10_000)  # shuffles the shards order a
dataloader = torch.utils.data.DataLoader(iterable_dataset, num_workers=4)  # assigns 64 / 4 = 16 s
```

Shuffle

Like a regular Dataset object, you can also shuffle a IterableDataset with IterableDataset.shuffle().

The `buffer_size` argument controls the size of the buffer to randomly sample examples from. Let's say your dataset has one million examples, and you set the `buffer_size` to ten thousand. IterableDataset.shuffle() will randomly select examples from the first ten thousand examples in the buffer. Selected examples in the buffer are replaced with new examples. By default, the buffer size is 1,000.

```
>>> from datasets import load_dataset
>>> dataset = load_dataset('HuggingFaceFW/fineweb', split='train', streaming=True)
>>> shuffled_dataset = dataset.shuffle(seed=42, buffer_size=10_000)
```

> [!TIP]
> IterableDataset.shuffle() will also shuffle the order of the shards if the dataset is sharded into multiple files.

Reshuffle

Sometimes you may want to reshuffle the dataset after each epoch. This will require you to set a different seed for each epoch. Use `IterableDataset.set_epoch()` in between epochs to tell the dataset what epoch you're on.

Your seed effectively becomes: `initial seed + current epoch`.

```
>>> for epoch in range(epochs):
...     shuffled_dataset.set_epoch(epoch)
...     for example in shuffled_dataset:
...         ...
```

Split dataset

You can split your dataset one of two ways:

- IterableDataset.take() returns the first `n` examples in a dataset:

```
>>> dataset = load_dataset('HuggingFaceFW/fineweb', split='train', streaming=True)
>>> dataset_head = dataset.take(2)
>>> list(dataset_head)
[{'text': "How AP reported in all formats from tor...},
 {'text': 'Did you know you have two little yellow...}]
```

- IterableDataset.skip() omits the first `n` examples in a dataset and returns the remaining examples:

```
>>> train_dataset = shuffled_dataset.skip(1000)
```

> [!WARNING]

> `take` and `skip` prevent future calls to `shuffle` because they lock in the order of the shards. You should `shuffle` your dataset before splitting it.

Shard

🤗 Datasets supports sharding to divide a very large dataset into a predefined number of chunks. Specify the `num_shards` parameter in shard() to determine the number of shards to split the dataset into. You'll also need to provide the shard you want to return with the `index` parameter.

For example, the amazon_polarity dataset has 4 shards (in this case they are 4 Parquet files):

```
>>> from datasets import load_dataset
>>> dataset = load_dataset("amazon_polarity", split="train", streaming=True)
>>> print(dataset)
IterableDataset({
    features: ['label', 'title', 'content'],
    num_shards: 4
})
```

After sharding the dataset into two chunks, the first one will only have 2 shards:

```
>>> dataset.shard(num_shards=2, index=0)
IterableDataset({
    features: ['label', 'title', 'content'],
    num_shards: 2
})
```

If your dataset has `dataset.num_shards==1`, you should chunk it using IterableDataset.skip() and IterableDataset.take() instead.

Interleave

interleave_datasets() can combine an IterableDataset with other datasets. The combined dataset returns alternating examples from each of the original datasets.

```
>>> from datasets import interleave_datasets
>>> es_dataset = load_dataset('allenai/c4', 'es', split='train', streaming=True)
>>> fr_dataset = load_dataset('allenai/c4', 'fr', split='train', streaming=True)

>>> multilingual_dataset = interleave_datasets([es_dataset, fr_dataset])
>>> list(multilingual_dataset.take(2))
[{'text': 'Comprar Zapatillas para niña en chancla con goma por...'},
 {'text': 'Le sacre de philippe ier, 23 mai 1059 - Compte Rendu...'}]
```

Define sampling probabilities from each of the original datasets for more control over how each of them are sampled and combined. Set the `probabilities` argument with your desired sampling probabilities:

```
>>> multilingual_dataset_with_oversampling = interleave_datasets([es_dataset, fr_dataset], probabi
>>> list(multilingual_dataset_with_oversampling.take(2))
[{'text': 'Comprar Zapatillas para niña en chancla con goma por...'},
 {'text': 'Chevrolet Cavalier Usados en Bogota - Carros en Vent...'}]
```

Around 80% of the final dataset is made of the `es_dataset`, and 20% of the `fr_dataset`.

You can also specify the `stopping_strategy`. The default strategy, `first_exhausted`, is a subsampling strategy, i.e the dataset construction is stopped as soon one of the dataset runs out of samples.
You can specify `stopping_strategy=all_exhausted` to execute an oversampling strategy. In this case, the dataset construction is stopped as soon as every samples in every dataset has been added at least once. In practice, it means that if a dataset is exhausted, it will return to the beginning of this dataset until the stop criterion has been reached.
Note that if no sampling probabilities are specified, the new dataset will have `max_length_datasets*nb_dataset samples`.
There is also `stopping_strategy=all_exhausted_without_replacement` to ensure that every sample is seen exactly once.

Rename, remove, and cast

The following methods allow you to modify the columns of a dataset. These methods are useful for renaming or removing columns and changing columns to a new set of features.

Rename

Use IterableDataset.rename_column() when you need to rename a column in your dataset. Features associated with the original column are actually moved under the new column name, instead of just replacing the original column in-place.

Provide IterableDataset.rename_column() with the name of the original column, and the new column name:

```
>>> from datasets import load_dataset
>>> dataset = load_dataset('allenai/c4', 'en', streaming=True, split='train')
>>> dataset = dataset.rename_column("text", "content")
```

Remove

When you need to remove one or more columns, give IterableDataset.remove_columns() the name of the column to remove. Remove more than one column by providing a list of column names:

```
>>> from datasets import load_dataset
>>> dataset = load_dataset('allenai/c4', 'en', streaming=True, split='train')
>>> dataset = dataset.remove_columns('timestamp')
```

Cast

IterableDataset.cast() changes the feature type of one or more columns. This method takes your new `Features` as its argument. The following sample code shows how to change the feature types of `ClassLabel` and `Value`:

```
>>> from datasets import load_dataset
>>> dataset = load_dataset('nyu-mll/glue', 'mrpc', split='train', streaming=True)
>>> dataset.features
{'sentence1': Value('string'),
'sentence2': Value('string'),
'label': ClassLabel(names=['not_equivalent', 'equivalent']),
'idx': Value('int32')}

>>> from datasets import ClassLabel, Value
>>> new_features = dataset.features.copy()
>>> new_features["label"] = ClassLabel(names=['negative', 'positive'])
>>> new_features["idx"] = Value('int64')
>>> dataset = dataset.cast(new_features)
>>> dataset.features
{'sentence1': Value('string'),
'sentence2': Value('string'),
'label': ClassLabel(names=['negative', 'positive']),
'idx': Value('int64')}
```

> [!TIP]
> Casting only works if the original feature type and new feature type are compatible. For example, you can cast a column with the feature type `Value('int32')` to `Value('bool')` if the original column only contains ones and zeros.

Use IterableDataset.cast_column() to change the feature type of just one column. Pass the column name and its new feature type as arguments:

```
>>> dataset.features
{'audio': Audio(sampling_rate=44100, mono=True)}

>>> dataset = dataset.cast_column("audio", Audio(sampling_rate=16000))
>>> dataset.features
{'audio': Audio(sampling_rate=16000, mono=True)}
```

Map

Similar to the Dataset.map() function for a regular Dataset, 🤗 Datasets features

IterableDataset.map() for processing an IterableDataset.
IterableDataset.map() applies processing on-the-fly when examples are streamed.

It allows you to apply a processing function to each example in a dataset, independently or in batches. This function can even create new rows and columns.

The following example demonstrates how to tokenize a IterableDataset. The function needs to accept and output a `dict`:

```
>>> def add_prefix(example):
...     example['text'] = 'My text: ' + example['text']
...     return example
```

Next, apply this function to the dataset with IterableDataset.map():

```
>>> from datasets import load_dataset
>>> dataset = load_dataset('allenai/c4', 'en', streaming=True, split='train')
>>> updated_dataset = dataset.map(add_prefix)
>>> list(updated_dataset.take(3))
[{'text': 'My text: Beginners BBQ Class Taking Place in Missoula!\nDo you want to get better at ma
  'timestamp': '2019-04-25 12:57:54',
  'url': 'https://klyq.com/beginners-bbq-class-taking-place-in-missoula/'},
 {'text': 'My text: Discussion in \'Mac OS X Lion (10.7)\' started by axboi87, Jan 20, 2012.\nI\'v
  'timestamp': '2019-04-21 10:07:13',
  'url': 'https://forums.macrumors.com/threads/restore-from-larger-disk-to-smaller-disk.1311329/'}
 {'text': 'My text: Foil plaid lycra and spandex shortall with metallic slinky insets. Attached me
  'timestamp': '2019-04-25 10:40:23',
  'url': 'https://awishcometrue.com/Catalogs/Clearance/Tweens/V1960-Find-A-Way'}]
```

Let's take a look at another example, except this time, you will remove columns with IterableDataset.map(). When you remove a column, it is only removed after the example has been provided to the mapped function. This allows the mapped function to use the content of the columns before they are removed.

Specify the column to remove with the `remove_columns` argument in IterableDataset.map():

```
>>> updated_dataset = dataset.map(add_prefix, remove_columns=["timestamp", "url"])
>>> list(updated_dataset.take(3))
[{'text': 'My text: Beginners BBQ Class Taking Place in Missoula!\nDo you want to get better at ma
 {'text': 'My text: Discussion in \'Mac OS X Lion (10.7)\' started by axboi87, Jan 20, 2012.\nI\'v
 {'text': 'My text: Foil plaid lycra and spandex shortall with metallic slinky insets. Attached me
```

Batch processing

IterableDataset.map() also supports working with batches of examples. Operate on batches by setting `batched=True`. The default batch size is 1000, but you can adjust it with the `batch_size` argument. This opens the door to many interesting applications such as tokenization, splitting long sentences into shorter chunks, and data augmentation.

Tokenization

```
>>> from datasets import load_dataset
>>> from transformers import AutoTokenizer
>>> dataset = load_dataset("allenai/c4", "en", streaming=True, split="train")
>>> tokenizer = AutoTokenizer.from_pretrained('distilbert-base-uncased')
>>> def encode(examples):
...     return tokenizer(examples['text'], truncation=True, padding='max_length')
>>> dataset = dataset.map(encode, batched=True, remove_columns=["text", "timestamp", "url"])
>>> next(iter(dataset))
{'input_ids': [101, 4088, 16912, 22861, 4160, 2465, 2635, 2173, 1999, 3335, ..., 0, 0, 0],
'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ..., 0, 0]}
```

> [!TIP]
> See other examples of batch processing in the batched map processing documentation. They work the same for iterable datasets.

Filter

You can filter rows in the dataset based on a predicate function using Dataset.filter(). It returns rows that match a specified condition:

```python
>>> from datasets import load_dataset
>>> dataset = load_dataset('HuggingFaceFW/fineweb', streaming=True, split='train')
>>> start_with_ar = dataset.filter(lambda example: example['text'].startswith('San Francisco'))
>>> next(iter(start_with_ar))
{'text': 'San Francisco 49ers cornerback Shawntae Spencer will miss the rest of the sea...}
```

Dataset.filter() can also filter by indices if you set `with_indices=True`:

```python
>>> even_dataset = dataset.filter(lambda example, idx: idx % 2 == 0, with_indices=True)
>>> list(even_dataset.take(3))
[{'text': 'How AP reported in all formats from tornado-stricken regionsMarch 8, 2012 Whe...},
 {'text': 'Car Wash For Clara! Now is your chance to help! 2 year old Clara Woodward has...},
 {'text': 'Log In Please enter your ECode to log in. Forgotten your eCode? If you create...}]
```

Batch

The `batch` method transforms your `IterableDataset` into an iterable of batches. This is particularly useful when you want to work with batches in your training loop or when using frameworks that expect batched inputs.

> [!TIP]
> There is also a "Batch Processing" option when using the `map` function to apply a function to batches of data, which is discussed in the Map section above. The `batch` method described here is different and provides a more direct way to create batches from your dataset.

You can use the `batch` method like this:

```python
from datasets import load_dataset

# Load a dataset in streaming mode
dataset = load_dataset("some_dataset", split="train", streaming=True)

# Create batches of 32 samples
batched_dataset = dataset.batch(batch_size=32)

# Iterate over the batched dataset
for batch in batched_dataset:
    print(batch)
    break
```

In this example, batched_dataset is still an IterableDataset, but each item yielded is now a batch of 32 samples instead of a single sample.
This batching is done on-the-fly as you iterate over the dataset, preserving the memory-efficient nature of IterableDataset.

The batch method also provides a drop_last_batch parameter.
When set to True, it will discard the last batch if it's smaller than the specified batch_size.
This can be useful in scenarios where your downstream processing requires all batches to be of the same size:

```python
batched_dataset = dataset.batch(batch_size=32, drop_last_batch=True)
```

Stream in a training loop

IterableDataset can be integrated into a training loop. First, shuffle the dataset:

```py >>> seed, buffer_size = 42, 10_000 >>> dataset = dataset.shuffle(seed, buffer_size=buffer_size) ```
Lastly, create a simple training loop and start training:
```

```
>>> import torch
>>> from torch.utils.data import DataLoader
>>> from transformers import AutoModelForMaskedLM, DataCollatorForLanguageModeling
>>> from tqdm import tqdm
>>> dataset = dataset.with_format("torch")
>>> dataloader = DataLoader(dataset, collate_fn=DataCollatorForLanguageModeling(tokenizer))
>>> device = 'cuda' if torch.cuda.is_available() else 'cpu'
>>> model = AutoModelForMaskedLM.from_pretrained("distilbert-base-uncased")
>>> model.train().to(device)
>>> optimizer = torch.optim.AdamW(params=model.parameters(), lr=1e-5)
>>> for epoch in range(3):
...     dataset.set_epoch(epoch)
...     for i, batch in enumerate(tqdm(dataloader, total=5)):
...         if i == 5:
...             break
...         batch = {k: v.to(device) for k, v in batch.items()}
...         outputs = model(**batch)
...         loss = outputs[0]
...         loss.backward()
...         optimizer.step()
...         optimizer.zero_grad()
...         if i % 10 == 0:
...             print(f"loss: {loss}")
```

Save a dataset checkpoint and resume iteration

If your training loop stops, you may want to restart the training from where it was. To do so you can save a checkpoint of your model and optimizers, as well as your data loader.

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)
>>> for idx, example in enumerate(iterable_dataset):
...     print(example)
...     if idx == 2:
...         state_dict = iterable_dataset.state_dict()
...         print("checkpoint")
...         break
>>> iterable_dataset.load_state_dict(state_dict)
>>> print(f"restart from checkpoint")
>>> for example in iterable_dataset:
...     print(example)
```

Returns:

```
{'a': 0}
{'a': 1}
{'a': 2}
checkpoint
restart from checkpoint
{'a': 3}
{'a': 4}
{'a': 5}
```

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`:

```
>>> from torchdata.stateful_dataloader import StatefulDataLoader
>>> iterable_dataset = load_dataset("deepmind/code_contests", streaming=True, split="train")
>>> dataloader = StatefulDataLoader(iterable_dataset, batch_size=32, num_workers=4)
>>> # checkpoint
>>> state_dict = dataloader.state_dict()  # uses iterable_dataset.state_dict() under the hood
>>> # resume from checkpoint
>>> dataloader.load_state_dict(state_dict)  # uses iterable_dataset.load_state_dict() under the ho
```

> [!TIP]
> Resuming returns exactly where the checkpoint was saved except if `.shuffle()` is used: examples from shuffle buffers are lost when resuming and the buffers are refilled with new data.

Save

Once your iterable dataset is ready, you can save it as a Hugging Face Dataset in Parquet format and reuse it later with load_dataset().

Save your dataset by providing the name of the dataset repository on Hugging Face you wish to save it to to push_to_hub(). This iterates over the dataset and progressively uploads the data to Hugging Face:

```
dataset.push_to_hub("username/my_dataset")
```

If the dataset consists of multiple shards (`dataset.num_shards > 1`), you can use multiple processes to upload it in parallel. This is especially useful if you applied `map()` or `filter()` steps since they will run faster in parallel:

```
dataset.push_to_hub("username/my_dataset", num_proc=8)
```

Use the load_dataset() function to reload the dataset:

```
from datasets import load_dataset
reloaded_dataset = load_dataset("username/my_dataset")
```

Export

🤗 Datasets supports exporting as well so you can work with your dataset in other applications. The following table shows currently supported file formats you can export to:

File type	Export method
CSV	IterableDataset.to_csv()
JSON	IterableDataset.to_json()
Parquet	IterableDataset.to_parquet()
SQL	IterableDataset.to_sql()
In-memory Python object	IterableDataset.to_pandas(), `IterableDataset.to_polars()` or IterableDataset.to_dict()

For example, export your dataset to a CSV file like this:

```
>>> dataset.to_csv("path/of/my/dataset.csv")
```

If you have a large dataset, you can save one file per shard, e.g.

```
>>> num_shards = dataset.num_shards
>>> for index in range(num_shards):
...     shard = dataset.shard(index, num_shards)
...     shard.to_parquet(f"path/of/my/dataset/data-{index:05d}.parquet")
```