Sự khác biệt giữa Tập dữ liệu vàBộ dữ liệu có thể lặp lại

Có hai loại đối tượng tập dữ liệu, Tập dữ liệu và Tập dữ liệu lặp.
Việc bạn chọn sử dụng hoặc tạo loại tập dữ liệu nào đều tùy thuộc vào kích thước của tập dữ liệu.
In general, an IterableDataset is ideal for big datasets (think hundreds of GBs!) due to its lazy
lợi thế về hành vi và tốc độ, trong khi Tập dữ liệu lại tuyệt vời cho mọi thứ khác.
Trang này sẽ so sánh sự khác biệt giữa Tập dữ liệu và IterableDataset để giúp bạn
chọn đối tượng tập dữ liệu phù hợp với bạn.

Đang tải xuống và phát trực tuyến

When you have a regular Dataset, you can access it using my_dataset[0] . This provides
truy cập ngẫu nhiên vào các hàng.
Các bộ dữ liệu như vậy còn được gọi là bộ dữ liệu "kiểu bản đồ".
Ví dụ: bạn có thể tải xuống ImageNet-1k như thế này và truy cập bất kỳ hàng nào:

```
từ tập dữ liệu nhập Load_dataset

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

Nhưng một lưu ý là bạn phải lưu trữ toàn bộ tập dữ liệu trên đĩa hoặc trong bộ nhớ,
ngăn bạn truy cập các tập dữ liệu lớn hơn đĩa.
Bởi vì nó có thể trở nên bất tiện đối với các tập dữ liệu lớn nên tồn tại một loại tập dữ liệu khác,
IterableDataset.
Khi bạn có IterableDataset , bạn có thể truy cập nó bằng vòng lặp for để tải dữ liệu
dần dần khi bạn lặp lại tập dữ liệu.
Bằng cách này, chỉ một phần nhỏ các ví dụ được tải vào bộ nhớ và bạn không viết gì cả
trên đĩa.

Ví dụ: bạn có thể truyền trực tuyến tập dữ liệu ImageNet-1k mà không cần tải xuống đĩa:

từ tập dữ liệu nhập Load_dataset

```
imagenet = load_dataset("timm/imagenet-1k-wds", split="train", streaming=True)# will start loadi
ví dụ trong imagenet:
    print(example)
    phá vỡ
```

Truyền phát có thể đọc dữ liệu trực tuyến mà không cần ghi bất kỳ tệp nào vào đĩa.
Ví dụ: bạn có thể truyền phát các tập dữ liệu được tạo từ nhiều phân đoạn, mỗi phân đoạn có hàng trăm phâ
gigabyte như C4 hoặc LAION-2B.
Tìm hiểu thêm về cách truyền phát tập dữ liệu trong Hướng dẫn truyền phát tập dữ liệu.

Tuy nhiên, đây không phải là điểm khác biệt duy nhất vì hành vi "lười biếng" của IterableDataset là
cũng có mặt khi nói đến việc tạo và xử lý dữ liệu.

Tạo bộ dữ liệu kiểu bản đồ và bộ dữ liệu có thể lặp lại

Bạn có thể tạo Tập dữ liệu bằng danh sách hoặc từ điển và dữ liệu được chuyển đổi hoàn toàn thành Mũi tên
để bạn có thể dễ dàng truy cập bất kỳ hàng nào:

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

Mặt khác, để tạo một IterableDataset, bạn phải cung cấp một cách "lười" để tải
dữ liệu.
Trong Python, chúng ta thường sử dụng các hàm tạo. Các hàm này mang lại một ví dụ tại một
thời gian, điều đó có nghĩa là bạn không thể truy cập một hàng bằng cách cắt nó như một Dataset thông thu

```
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})
ví dụ trong my_iterable_dataset:
    print(example)
    phá vỡ
```

Đang tải các tập tin cục bộ hoàn toàn và dần dần

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])
```

Tuy nhiên, việc này yêu cầu một bước chuyển đổi từ định dạng CSV sang định dạng Mũi tên, tốn thời gian và không gian nếu tập dữ liệu của bạn lớn.

Để tiết kiệm dung lượng ổ đĩa và bỏ qua bước chuyển đổi, bạn có thể xác định IterableDataset bằng cách phát trực tiếp từ các tập tin cục bộ.
Bằng cách này, dữ liệu được đọc dần dần từ các tệp cục bộ khi bạn lặp lại tập dữ liệu:

```
data_files = {"train": ["path/to/data.csv"]}
my_iterable_dataset = load_dataset("csv", data_files=data_files, split="train", streaming=True)
ví dụ trong my_iterable_dataset:# cái này đọc tệp CSV dần dần khi bạn lặp lại t
    print(example)
    phá vỡ
```

Nhiều định dạng tệp được hỗ trợ, như CSV, JSONL và Parquet, cũng như hình ảnh và âm thanh tập tin.
Bạn có thể tìm thêm thông tin trong các hướng dẫn tương ứng để tải dạng bảng, văn bản, hình ảnh và bộ dữ liệu âm thanh.

Xử lý dữ liệu háo hức và xử lý dữ liệu lười biếng

When you process a Dataset object using Dataset.map(), the entire dataset is processed ngay lập tức và quay trở lại.
Điều này tương tự như cách hoạt động của gấu trúc chẳng hạn.

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

Mặt khác, do tính chất "lười biếng" của IterableDataset nên việc gọi
IterableDataset.map() does not apply your map function over the full dataset.
Thay vào đó, chức năng bản đồ của bạn được áp dụng nhanh chóng.

Do đó, bạn có thể xâu chuỗi nhiều bước xử lý và tất cả chúng sẽ chạy cùng một lúc khi
bạn bắt đầu lặp lại tập dữ liệu:

```
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 và process_fn_2 được áp dụng nhanh chóng khi lặp qua tập dữ liệu
ví dụ trong my_iterable_dataset:
    print(example)
    phá vỡ
```

Xáo trộn gần đúng chính xác và nhanh chóng

When you shuffle a Dataset using Dataset.shuffle(), you apply an exact shuffling of the
tập dữ liệu.
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
ánh xạ chỉ số đã được xáo trộn:

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

Vì chúng tôi không có quyền truy cập ngẫu nhiên vào các hàng trong trường hợp IterableDataset nên chúng
sử dụng danh sách chỉ mục được xáo trộn và truy cập một hàng ở vị trí tùy ý.
Điều này ngăn cản việc sử dụng xáo trộn chính xác.
Instead, a fast approximate shuffling is used in IterableDataset.shuffle().
Nó sử dụng bộ đệm xáo trộn để lấy mẫu ngẫu nhiên lặp đi lặp lại từ tập dữ liệu.
Vì tập dữ liệu vẫn được đọc lặp đi lặp lại nên nó mang lại hiệu suất tốc độ tuyệt vời:

```
my_iterable_dataset = my_iterable_dataset.shuffle(seed=42, buffer_size=100)
ví dụ trong my_iterable_dataset:
    print(example)
    phá vỡ
```

Nhưng sử dụng bộ đệm xáo trộn là không đủ để mang lại sự xáo trộn thỏa đáng cho máy học model training. So IterableDataset.shuffle() also shuffles the dataset shards if your dataset is được tạo từ nhiều tệp hoặc nguồn:

```
# Truyền phát từ internet
my_iterable_dataset = load_dataset("deepmind/code_contests", split="train", streaming=True)
my_iterable_dataset.num_shards# 39

# Truyền phát từ các tệp cục bộ
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

# Từ hàm tạo
def my_generator(n, sources):
    cho nguồn trong các nguồn:
        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
```

Chênh lệch tốc độ

Các đối tượng Bộ dữ liệu thông thường dựa trên Mũi tên cung cấp khả năng truy cập ngẫu nhiên nhanh chó
Nhờ ánh xạ bộ nhớ và thực tế là Arrow là định dạng trong bộ nhớ, đọc dữ liệu từ
disk không thực hiện các cuộc gọi hệ thống và giải tuần tự hóa đắt tiền.
Nó thậm chí còn cung cấp khả năng tải dữ liệu nhanh hơn khi lặp bằng vòng lặp for bằng cách lặp trên các p
Mũi tên ghi lại lô.

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

tốc độ có thể chậm hơn gấp 10 lần.
Điều này là do có thêm một bước để đọc chỉ mục hàng bằng cách sử dụng ánh xạ chỉ mục,
và quan trọng nhất, bạn không còn đọc những khối dữ liệu liền kề nhau nữa.
Để khôi phục tốc độ, bạn cần phải ghi lại toàn bộ tập dữ liệu trên đĩa của mình bằng cách sử dụng
Dataset.flatten_indices(), which removes the indices mapping.
Tuy nhiên, việc này có thể mất nhiều thời gian tùy thuộc vào kích thước tập dữ liệu của bạn:

```
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
```

Trong trường hợp này, chúng tôi khuyên bạn nên chuyển sang IterableDataset và tận dụng tốc độ nhanh củ
approximate shuffling method IterableDataset.shuffle().
Nó chỉ xáo trộn thứ tự phân đoạn và thêm bộ đệm ngẫu nhiên vào tập dữ liệu của bạn, giúp giữ nguyên
tốc độ tối ưu của tập dữ liệu của bạn.
Bạn cũng có thể cải tổ lại tập dữ liệu một cách dễ dàng:

```
for example in enumerate(my_iterable_dataset):# fast
    vượt qua

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

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

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
    vượt qua
```

Nếu bạn đang sử dụng tập dữ liệu của mình trên nhiều kỷ nguyên, hạt giống hiệu quả sẽ xáo trộn thứ tự phâ
bộ đệm xáo trộn là Seed + Epoch .
Nó giúp dễ dàng sắp xếp lại tập dữ liệu giữa các kỷ nguyên:

```
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 =
        vượt qua
```

Để bắt đầu lại quá trình lặp của tập dữ liệu kiểu bản đồ, bạn chỉ cần bỏ qua các ví dụ đầu tiên:

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

Nhưng nếu bạn sử dụng DataLoader với Sampler , thay vào đó bạn nên lưu trạng thái của mình
sampler (you might have written a custom sampler that allows resuming).

Mặt khác, các bộ dữ liệu có thể lặp lại không cung cấp quyền truy cập ngẫu nhiên vào một chỉ mục mẫu cụ th
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
có thể làm cho các mô hình và trình tối ưu hóa:

```
>>> 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)
```

Dưới lớp vỏ bọc, tập dữ liệu có thể lặp lại theo dõi phân đoạn hiện tại đang được đọc và
chỉ mục ví dụ trong phân đoạn hiện tại và nó lưu trữ thông tin này trong state_dict .

Để tiếp tục từ điểm kiểm tra, tập dữ liệu sẽ bỏ qua tất cả các phân đoạn đã được đọc trước đó
khởi động lại từ phân đoạn hiện tại.
Sau đó, nó đọc phân đoạn và bỏ qua các ví dụ cho đến khi đạt được ví dụ chính xác từ
trạm kiểm soát.

Do đó, việc khởi động lại tập dữ liệu khá nhanh vì nó sẽ không đọc lại các phân đoạn đã
đã được lặp đi lặp lại. Tuy nhiên, việc tiếp tục lại tập dữ liệu thường không phải là ngay lập tức vì nó phải khở
đọc từ đầu phân đoạn hiện tại và bỏ qua các ví dụ cho đến khi đạt đến phần
vị trí trạm kiểm soát.

Điều này có thể được sử dụng với StatefulDataLoader từ torchdata, xem phát trực tuyến bằng PyTorch

Trình tải dữ liệu.

Chuyển từ kiểu bản đồ sang kiểu lặp

Nếu bạn muốn hưởng lợi từ hành vi "lười biếng" của IterableDataset hoặc lợi thế về tốc độ của chúng, bạn có thể chuyển Bộ dữ liệu kiểu bản đồ của mình thành IterableDataset:

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

Nếu bạn muốn xáo trộn tập dữ liệu của mình hoặc sử dụng nó với PyTorch DataLoader, chúng tôi khuyên bạ tạo một IterableDataset được phân chia:

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