Batch mapping

Combining the utility of Dataset.map() with batch mode is very powerful. It allows you to speed up processing, and freely control the size of the generated dataset.

Need for speed

The primary objective of batch mapping is to speed up processing. Often times, it is faster to work with batches of data instead of single examples. Naturally, batch mapping lends itself to tokenization. For example, the 🤗 Tokenizers library works faster with batches because it parallelizes the tokenization of all the examples in a batch.

Input size != output size

The ability to control the size of the generated dataset can be leveraged for many interesting use-cases. In the How-to map section, there are examples of using batch mapping to:

- Split long sentences into shorter chunks.
- Augment a dataset with additional tokens.

It is helpful to understand how this works, so you can come up with your own ways to use batch mapping. At this point, you may be wondering how you can control the size of the generated dataset. The answer is: **the mapped function does not have to return an output batch of the same size**.

In other words, your mapped function input can be a batch of size `N` and return a batch of size `M`. The output `M` can be greater than or less than `N`. This means you can concatenate your examples, divide it up, and even add more examples!

However, remember that all values in the output dictionary must contain the **same number of elements** as the other fields in the output dictionary. Otherwise, it is not possible to define the number of examples in the output returned by the mapped function. The number can vary between successive batches processed by the mapped function. For a single batch though, all values of the output dictionary should have the same length (i.e., the number of elements).

For example, from a dataset of 1 column and 3 rows, if you use `map` to return a new column

with twice as many rows, then you will have an error.
In this case, you end up with one column with 3 rows, and one column with 6 rows. As you can see, the table will not be valid:

```
>>> from datasets import Dataset
>>> dataset = Dataset.from_dict({"a": [0, 1, 2]})
>>> dataset.map(lambda batch: {"b": batch["a"] * 2}, batched=True)  # new column with 6 elements:
'ArrowInvalid: Column 1 named b expected length 3 but got length 6'
```

To make it valid, you have to drop one of the columns:

```
>>> from datasets import Dataset
>>> dataset = Dataset.from_dict({"a": [0, 1, 2]})
>>> dataset_with_duplicates = dataset.map(lambda batch: {"b": batch["a"] * 2}, remove_columns=["a"
>>> len(dataset_with_duplicates)
6
```

Alternatively, you can overwrite the existing column to achieve the same result.
For example, here's how to duplicate every row in the dataset by overwriting column `"a"`:

```
>>> from datasets import Dataset
>>> dataset = Dataset.from_dict({"a": [0, 1, 2]})
# overwrites the existing "a" column with duplicated values
>>> duplicated_dataset = dataset.map(
...     lambda batch: {"a": [x for x in batch["a"] for _ in range(2)]},
...     batched=True
... )
>>> duplicated_dataset
Dataset({
    features: ['a'],
    num_rows: 6
})
>>> duplicated_dataset["a"]
[0, 0, 1, 1, 2, 2]
```