Using Datasets with TensorFlow

This document is a quick introduction to using `datasets` with TensorFlow, with a particular focus on how to get
`tf.Tensor` objects out of our datasets, and how to stream data from Hugging Face `Dataset` objects to Keras methods
like `model.fit()`.

Dataset format

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

To get TensorFlow tensors instead, you can set the format of the dataset to `tf`:

```
>>> from datasets import Dataset
>>> data = [[1, 2],[3, 4]]
>>> ds = Dataset.from_dict({"data": data})
>>> ds = ds.with_format("tf")
>>> ds[0]
{'data': <tf.Tensor: shape=(2,), dtype=int64, numpy=array([1, 2])>}
>>> ds[:2]
{'data': <tf.Tensor: shape=(2, 2), dtype=int64, numpy=
array([[1, 2],
       [3, 4]])>}
```

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

This can be useful for converting your dataset to a dict of `Tensor` objects, or for writing a generator to load TF
samples from it. If you wish to convert the entire dataset to `Tensor`, simply query the full dataset:

```
>>> ds[:]
{'data': <tf.Tensor: shape=(2, 2), dtype=int64, numpy=
array([[1, 2],
       [3, 4]])>}
```

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("tf")
>>> ds[0]
{'data': <tf.Tensor: shape=(2, 2), dtype=int64, numpy=
 array([[1, 2],
        [3, 4]])>}
```

Otherwise, a TensorFlow formatted dataset outputs a `RaggedTensor` instead of a single tensor:

```
>>> 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': <tf.RaggedTensor [[1, 2], [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("tf")
>>> ds[0]
{'data': <tf.Tensor: shape=(2, 2), dtype=int64, numpy=
 array([[1, 2],
        [3, 4]])>}
>>> ds[:2]
{'data': <tf.Tensor: shape=(2, 2, 2), dtype=int64, numpy=
 array([[[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("tf")
>>> ds[:3]
{'label': <tf.Tensor: shape=(3,), dtype=int64, numpy=array([0, 0, 1])>}
```

Strings and binary objects are also supported:

```
>>> from datasets import Dataset, Features
>>> text = ["foo", "bar"]
>>> data = [0, 1]
>>> ds = Dataset.from_dict({"text": text, "data": data})
>>> ds = ds.with_format("tf")
>>> ds[:2]
{'text': <tf.Tensor: shape=(2,), dtype=string, numpy=array([b'foo', b'bar'], dtype=object)>,
 'data': <tf.Tensor: shape=(2,), dtype=int64, numpy=array([0, 1])>}
```

You can also explicitly format certain columns and leave the other columns unformatted:

```
>>> ds = ds.with_format("tf", columns=["data"], output_all_columns=True)
>>> ds[:2]
{'data': <tf.Tensor: shape=(2,), dtype=int64, numpy=array([0, 1])>,
 'text': ['foo', 'bar']}
```

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("tf")
>>> ds[0]
{'image': <tf.Tensor: shape=(512, 512, 4), dtype=uint8, numpy=
 array([[[255, 215, 106, 255],
         [255, 215, 106, 255],
         ...,
         [255, 255, 255, 255],
         [255, 255, 255, 255]]], dtype=uint8)>}
>>> ds[:2]
{'image': <tf.Tensor: shape=(2, 512, 512, 4), dtype=uint8, numpy=
 array([[[[255, 215, 106, 255],
          [255, 215, 106, 255],
          ...,
          [255, 255, 255, 255],
          [255, 255, 255, 255]]]], dtype=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("tf")
>>> ds[0]["audio"]["array"]
<tf.Tensor: shape=(202311,), dtype=float32, numpy=
array([ 6.1035156e-05,  1.5258789e-05,  1.6784668e-04, ...,
       -1.5258789e-05, -1.5258789e-05,  1.5258789e-05], dtype=float32)>
>>> ds[0]["audio"]["sampling_rate"]
<tf.Tensor: shape=(), dtype=int32, numpy=44100>
```

Data loading

Although you can load individual samples and batches just by indexing into your dataset, this won't work if you want
to use Keras methods like `fit()` and `predict()`. You could write a generator function that shuffles and loads batches
from your dataset and `fit()` on that, but that sounds like a lot of unnecessary work. Instead, if you want to stream
data from your dataset on-the-fly, we recommend converting your dataset to a `tf.data.Dataset` using the
`to_tf_dataset()` method.

The `tf.data.Dataset` class covers a wide range of use-cases - it is often created from Tensors in memory, or using a load function to read files on disc
or external storage. The dataset can be transformed arbitrarily with the `map()` method, or methods like `batch()`
and `shuffle()` can be used to create a dataset that's ready for training. These methods do not modify the stored data
in any way - instead, the methods build a data pipeline graph that will be executed when the dataset is iterated over,
usually during model training or inference. This is different from the `map()` method of Hugging Face `Dataset` objects,
which runs the map function immediately and saves the new or changed columns.

Since the entire data preprocessing pipeline can be compiled in a `tf.data.Dataset`, this approach allows for massively
parallel, asynchronous data loading and training. However, the requirement for graph compilation can be a limitation,
particularly for Hugging Face tokenizers, which are usually not (yet!) compilable as part of a TF graph. As a result,
we usually advise pre-processing the dataset as a Hugging Face dataset, where arbitrary Python functions can be
used, and then converting to `tf.data.Dataset` afterwards using `to_tf_dataset()` to get a batched dataset ready for
training. To see examples of this approach, please see the examples or notebooks for

`transformers`.

Using `to_tf_dataset()`

Using `to_tf_dataset()` is straightforward. Once your dataset is preprocessed and ready, simply call it like so:

```
>>> from datasets import Dataset
>>> data = {"inputs": [[1, 2],[3, 4]], "labels": [0, 1]}
>>> ds = Dataset.from_dict(data)
>>> tf_ds = ds.to_tf_dataset(
            columns=["inputs"],
            label_cols=["labels"],
            batch_size=2,
            shuffle=True
            )
```

The returned `tf_ds` object here is now fully ready to train on, and can be passed directly to `model.fit()`. Note
that you set the batch size when creating the dataset, and so you don't need to specify it when calling `fit()`:

```
>>> model.fit(tf_ds, epochs=2)
```

For a full description of the arguments, please see the to_tf_dataset() documentation. In many cases,
you will also need to add a `collate_fn` to your call. This is a function that takes multiple elements of the dataset
and combines them into a single batch. When all elements have the same length, the built-in default collator will
suffice, but for more complex tasks a custom collator may be necessary. In particular, many tasks have samples
with varying sequence lengths which will require a data collator that can pad batches correctly. You can see examples
of this in the `transformers` NLP examples and
notebooks, where variable sequence lengths are very common.

If you find that loading with `to_tf_dataset` is slow, you can also use the `num_workers` argument. This spins
up multiple subprocesses to load data in parallel. This feature is recent and still somewhat experimental - please file
an issue if you encounter any bugs while using it!

When to use to_tf_dataset

The astute reader may have noticed at this point that we have offered two approaches to achieve the same goal - if you
want to pass your dataset to a TensorFlow model, you can either convert the dataset to a `Tensor` or `dict` of `Tensors`
using `.with_format('tf')`, or you can convert the dataset to a `tf.data.Dataset` with `to_tf_dataset()`. Either of these
can be passed to `model.fit()`, so which should you choose?

The key thing to recognize is that when you convert the whole dataset to `Tensor`s, it is static and fully loaded into
RAM. This is simple and convenient, but if any of the following apply, you should probably use `to_tf_dataset()`
instead:

- Your dataset is too large to fit in RAM. `to_tf_dataset()` streams only one batch at a time, so even very large
  datasets can be handled with this method.
- You want to apply random transformations using `dataset.with_transform()` or the `collate_fn`. This is
  common in several modalities, such as image augmentations when training vision models, or random masking when training
  masked language models. Using `to_tf_dataset()` will apply those transformations
  at the moment when a batch is loaded, which means the same samples will get different augmentations each time
  they are loaded. This is usually what you want.
- Your data has a variable dimension, such as input texts in NLP that consist of varying numbers of tokens. When you create a batch with samples with a variable dimension, the standard solution is to

pad the shorter samples to the length of the longest one. When you stream samples from a dataset with `to_tf_dataset`,
you can apply this padding to each batch via your `collate_fn`. However, if you want to convert
such a dataset to dense `Tensor`s, then you will have to pad samples to the length of the longest sample in *the
entire dataset!* This can result in huge amounts of padding, which wastes memory and reduces your model's speed.

Caveats and limitations

Right now, `to_tf_dataset()` always returns a batched dataset - we will add support for unbatched datasets soon!