Use with JAX

This document is a quick introduction to using `datasets` with JAX, with a particular focus on how to get
`jax.Array` objects out of our datasets, and how to use them to train JAX models.

> [!TIP]
> `jax` and `jaxlib` are required to reproduce to code above, so please make sure you install them as `pip install datasets[jax]`.

Dataset format

By default, datasets return regular Python objects: integers, floats, strings, lists, etc., and string and binary objects are unchanged, since JAX only supports numbers.

To get JAX arrays (numpy-like) instead, you can set the format of the dataset to `jax`:

```
>>> from datasets import Dataset
>>> data = [[1, 2], [3, 4]]
>>> ds = Dataset.from_dict({"data": data})
>>> ds = ds.with_format("jax")
>>> ds[0]
{'data': DeviceArray([1, 2], dtype=int32)}
>>> ds[:2]
{'data': DeviceArray([
    [1, 2],
    [3, 4]], dtype=int32)}
```

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

Note that the exact same procedure applies to `DatasetDict` objects, so that when setting the format of a `DatasetDict` to `jax`, all the `Dataset`s there will be formatted as `jax`:

```
>>> from datasets import DatasetDict
>>> data = {"train": {"data": [[1, 2], [3, 4]]}, "test": {"data": [[5, 6], [7, 8]]}}
>>> dds = DatasetDict.from_dict(data)
>>> dds = dds.with_format("jax")
>>> dds["train"][:2]
{'data': DeviceArray([
    [1, 2],
    [3, 4]], dtype=int32)}
```

Another thing you'll need to take into consideration is that the formatting is not applied until you actually access the data. So if you want to get a JAX array out of a dataset, you'll need to access the data first, otherwise the format will remain the same.

Finally, to load the data in the device of your choice, you can specify the `device` argument, but note that `jaxlib.xla_extension.Device` is not supported as it's not serializable with neither `pickle` not `dill`, so you'll need to use its string identifier instead:

```
>>> import jax
>>> from datasets import Dataset
>>> data = [[1, 2], [3, 4]]
>>> ds = Dataset.from_dict({"data": data})
>>> device = str(jax.devices()[0])  # Not casting to `str` before passing it to `with_format` will
>>> ds = ds.with_format("jax", device=device)
>>> ds[0]
{'data': DeviceArray([1, 2], dtype=int32)}
>>> ds[0]["data"].device()
TFRT_CPU_0
>>> assert ds[0]["data"].device() == jax.devices()[0]
True
```

Note that if the `device` argument is not provided to `with_format` then it will use the default device which is `jax.devices()[0]`.

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("jax")
>>> ds[0]
{'data': Array([[1, 2],
        [3, 4]], dtype=int32)}
```

```
>>> 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("jax")
>>> ds[0]
{'data': [Array([1, 2], dtype=int32), Array([3], dtype=int32)]}
```

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("jax")
>>> ds[0]
{'data': Array([[1, 2],
        [3, 4]], dtype=int32)}
>>> ds[:2]
{'data': Array([[[1, 2],
         [3, 4]],

        [[5, 6],
         [7, 8]]], dtype=int32)}
```

Other feature types

ClassLabel data is properly converted to arrays:

```
>>> 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("jax")
>>> ds[:3]
{'label': DeviceArray([0, 0, 1], dtype=int32)}
```

String and binary objects are unchanged, since JAX 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, Image
>>> images = ["path/to/image.png"] * 10
>>> features = Features({"image": Image()})
>>> ds = Dataset.from_dict({"image": images}, features=features)
>>> ds = ds.with_format("jax")
>>> ds[0]["image"].shape
(512, 512, 3)
>>> ds[0]
{'image': DeviceArray([[[ 255, 255, 255],
              [ 255, 255, 255],
              ...,
              [ 255, 255, 255],
              [ 255, 255, 255]]], dtype=uint8)}
>>> ds[:2]["image"].shape
(2, 512, 512, 3)
>>> ds[:2]
{'image': DeviceArray([[[[ 255, 255, 255],
              [ 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
>>> audio = ["path/to/audio.wav"] * 10
>>> features = Features({"audio": Audio()})
>>> ds = Dataset.from_dict({"audio": audio}, features=features)
>>> ds = ds.with_format("jax")
>>> ds[0]["audio"]["array"]
DeviceArray([-0.059021  , -0.03894043, -0.00735474, ...,  0.0133667 ,
              0.01809692,  0.00268555], dtype=float32)
>>> ds[0]["audio"]["sampling_rate"]
DeviceArray(44100, dtype=int32, weak_type=True)
```

Data loading

JAX doesn't have any built-in data loading capabilities, so you'll need to use a library such as PyTorch to load your data using a `DataLoader` or TensorFlow using a `tf.data.Dataset`. Citing the JAX documentation on this topic:
"JAX is laser-focused on program transformations and accelerator-backed NumPy, so we don’t include data loading or munging in the JAX library. There are already a lot of great data loaders
out there, so let’s just use them instead of reinventing anything. We’ll grab PyTorch’s data loader,
and make a tiny shim to make it work with NumPy arrays.".

So that's the reason why JAX-formatting in `datasets` is so useful, because it lets you use any model from the HuggingFace Hub with JAX, without having to worry about the data loading
part.

Using `with_format('jax')`

The easiest way to get JAX arrays out of a dataset is to use the `with_format('jax')` method. Lets assume

that we want to train a neural network on the MNIST dataset available at the HuggingFace Hub at https://huggingface.co/datasets/mnist.

```
>>> from datasets import load_dataset
>>> ds = load_dataset("mnist")
>>> ds = ds.with_format("jax")
>>> ds["train"][0]
{'image': DeviceArray([[  0,   0,   0, ...],
                       [  0,   0,   0, ...],
                       ...,
                       [  0,   0,   0, ...],
                       [  0,   0,   0, ...]], dtype=uint8),
 'label': DeviceArray(5, dtype=int32)}
```

Once the format is set we can feed the dataset to the JAX model in batches using the `Dataset.iter()` method:

```
>>> for epoch in range(epochs):
...     for batch in ds["train"].iter(batch_size=32):
...         x, y = batch["image"], batch["label"]
...         ...
```