



Google I/O is a wrap! Catch up on TensorFlow sessions

View sessions (<https://io.google/2022/products/tensorflow/>)

tf.data: Build TensorFlow input pipelines


[Run in Google Colab](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/data.ipynb)


[View source on GitHub](https://github.com/tensorflow/tensorflow/blob/master/site/en/guide/data.ipynb)

The `tf.data` (https://www.tensorflow.org/api_docs/python/tf/data) API enables you to build complex input pipelines from simple, reusable pieces. For example, the pipeline for an image model might aggregate data from files in a distributed file system, apply random perturbations to each image, and merge randomly selected images into a batch for training. The pipeline for a text model might involve extracting symbols from raw text data, converting them to embedding identifiers with a lookup table, and batching together sequences of different lengths. The `tf.data` (https://www.tensorflow.org/api_docs/python/tf/data) API makes it possible to handle large amounts of data, read from different data formats, and perform complex transformations.

The `tf.data` (https://www.tensorflow.org/api_docs/python/tf/data) API introduces a `tf.data.Dataset` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset) abstraction that represents a sequence of elements, in which each element consists of one or more components. For example, in an image pipeline, an element might be a single training example, with a pair of tensor components representing the image and its label.

There are two distinct ways to create a dataset:

- A data **source** constructs a `Dataset` from data stored in memory or in one or more files.
- A data **transformation** constructs a dataset from one or more `tf.data.Dataset` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset) objects.

```
import tensorflow as tf
```

```
import pathlib
import os
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

np.set_printoptions(precision=4)
```

Basic mechanics

To create an input pipeline, you must start with a data *source*. For example, to construct a `Dataset` from data in memory, you can use `tf.data.Dataset.from_tensors()` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#from_tensors) or `tf.data.Dataset.from_tensor_slices()` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#from_tensor_slices). Alternatively, if your input data is stored in a

file in the recommended TFRecord format, you can use `tf.data.TFRecordDataset()` (https://www.tensorflow.org/api_docs/python/tf/data/TFRecordDataset).

Once you have a `Dataset` object, you can *transform* it into a new `Dataset` by chaining method calls on the `tf.data.Dataset` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset) object. For example, you can apply per-element transformations such as `Dataset.map` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#map), and multi-element transformations such as `Dataset.batch` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#batch). Refer to the documentation for `tf.data.Dataset` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset) for a complete list of transformations.

The `Dataset` object is a Python iterable. This makes it possible to consume its elements using a for loop:

```
dataset = tf.data.Dataset.from_tensor_slices([8, 3, 0, 8, 2, 1])
dataset
```

```
<TensorSliceDataset element_spec=TensorSpec(shape=(), dtype=tf.int32, name=None)>
```

```
for elem in dataset:
    print(elem.numpy())
```

```
8
3
0
8
2
1
```

Or by explicitly creating a Python iterator using `iter` and consuming its elements using `next`:

```
it = iter(dataset)
print(next(it).numpy())
```

```
8
```

Alternatively, dataset elements can be consumed using the `reduce` transformation, which reduces all elements to produce a single result. The following example illustrates how to use the `reduce` transformation to compute the sum of a dataset of integers.

```
print(dataset.reduce(0, lambda state, value: state + value).numpy())
```

```
22
```

Dataset structure

A dataset produces a sequence of *elements*, where each element is the same (nested) structure of *components*.

Individual components of the structure can be of any type representable by `tf.TypeSpec`

(https://www.tensorflow.org/api_docs/python/tf/TypeSpec), including `tf.Tensor`

(https://www.tensorflow.org/api_docs/python/tf/Tensor), `tf.sparse.SparseTensor`

(https://www.tensorflow.org/api_docs/python/tf/sparse/SparseTensor), `tf.RaggedTensor`

(https://www.tensorflow.org/api_docs/python/tf/RaggedTensor), `tf.TensorArray`

(https://www.tensorflow.org/api_docs/python/tf/TensorArray), or `tf.data.Dataset`

(https://www.tensorflow.org/api_docs/python/tf/data/Dataset).

The Python constructs that can be used to express the (nested) structure of elements include `tuple`, `dict`, `NamedTuple`, and `OrderedDict`. In particular, `list` is not a valid construct for expressing the structure of dataset elements. This is because early `tf.data` (https://www.tensorflow.org/api_docs/python/tf/data) users felt strongly about `list` inputs (for example, when passed to `tf.data.Dataset.from_tensors`

(https://www.tensorflow.org/api_docs/python/tf/data/Dataset#from_tensors)) being automatically packed as tensors and `list`

outputs (for example, return values of user-defined functions) being coerced into a `tuple`. As a consequence, if you

would like a `list` input to be treated as a structure, you need to convert it into `tuple` and if you would like a `list`

output to be a single component, then you need to explicitly pack it using `tf.stack`

(https://www.tensorflow.org/api_docs/python/tf/stack).

The `Dataset.element_spec` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#element_spec) property allows you to inspect the type of each element component. The property returns a *nested structure* of `tf.TypeSpec`

(https://www.tensorflow.org/api_docs/python/tf/TypeSpec) objects, matching the structure of the element, which may be a

single component, a tuple of components, or a nested tuple of components. For example:

```
dataset1 = tf.data.Dataset.from_tensor_slices(tf.random.uniform([4, 10]))
```

```
dataset1.element_spec
```

```
TensorSpec(shape=(10,), dtype=tf.float32, name=None)
```

```
dataset2 = tf.data.Dataset.from_tensor_slices(
    (tf.random.uniform([4]),
     tf.random.uniform([4, 100], maxval=100, dtype=tf.int32)))
```

```
dataset2.element_spec
```

```
(TensorSpec(shape=(), dtype=tf.float32, name=None),
 TensorSpec(shape=(100,), dtype=tf.int32, name=None))
```

```
dataset3 = tf.data.Dataset.zip((dataset1, dataset2))
```

```
dataset3.element_spec
```

```
(TensorSpec(shape=(10,), dtype=tf.float32, name=None),
 (TensorSpec(shape=(), dtype=tf.float32, name=None),
```

```
TensorSpec(shape=(100,), dtype=tf.int32, name=None)))
```

```
# Dataset containing a sparse tensor.
dataset4 = tf.data.Dataset.from_tensors(tf.SparseTensor(indices=[[0, 0], [1, 2]], values=[1, 2], dens
dataset4.element_spec
```

```
SparseTensorSpec(TensorShape([3, 4]), tf.int32)
```

```
# Use value_type to see the type of value represented by the element spec
dataset4.element_spec.value_type
```

```
tensorflow.python.framework.sparse_tensor.SparseTensor
```

The `Dataset` transformations support datasets of any structure. When using the `Dataset.map` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#map), and `Dataset.filter` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#filter) transformations, which apply a function to each element, the element structure determines the arguments of the function:

```
dataset1 = tf.data.Dataset.from_tensor_slices(
    tf.random.uniform([4, 10], minval=1, maxval=10, dtype=tf.int32))
dataset1
```

```
<TensorSliceDataset element_spec=TensorSpec(shape=(10,), dtype=tf.int32, name=None)>
```

```
for z in dataset1:
    print(z.numpy())
```

```
[5 2 8 8 3 7 9 3 2 2]
[1 6 4 2 3 5 4 4 8 9]
[6 3 3 6 2 9 2 4 1 8]
[3 6 5 1 6 5 7 4 8 1]
```

```
dataset2 = tf.data.Dataset.from_tensor_slices(
    (tf.random.uniform([4]),
     tf.random.uniform([4, 100], maxval=100, dtype=tf.int32)))
dataset2
```

```
<TensorSliceDataset element_spec=(TensorSpec(shape=(), dtype=tf.float32, name=None), TensorSpec(shape=
```

```
dataset3 = tf.data.Dataset.zip((dataset1, dataset2))
```

```
dataset3
```

```
<ZipDataset element_spec=(TensorSpec(shape=(10,), dtype=tf.int32, name=None), (TensorSpec(shape=(), c
```

```
for a, (b,c) in dataset3:
    print('shapes: {a.shape}, {b.shape}, {c.shape}'.format(a=a, b=b, c=c))
```

```
shapes: (10,), (), (100,)
shapes: (10,), (), (100,)
shapes: (10,), (), (100,)
shapes: (10,), (), (100,)
```

Reading input data

Consuming NumPy arrays

Refer to the [Loading NumPy arrays](https://www.tensorflow.org/tutorials/load_data/numpy) (https://www.tensorflow.org/tutorials/load_data/numpy) tutorial for more examples.

If all of your input data fits in memory, the simplest way to create a `Dataset` from them is to convert them to `tf.Tensor` (https://www.tensorflow.org/api_docs/python/tf/Tensor) objects and use `Dataset.from_tensor_slices` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#from_tensor_slices).

```
train, test = tf.keras.datasets.fashion_mnist.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-labels-idx1-ubyte.gz
29515/29515 [=====] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-images-idx3-ubyte.gz
26421880/26421880 [=====] - 1s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-labels-idx1-ubyte.gz
5148/5148 [=====] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-images-idx3-ubyte.gz
4422102/4422102 [=====] - 0s 0us/step
```

```
images, labels = train
images = images/255

dataset = tf.data.Dataset.from_tensor_slices((images, labels))
dataset
```

```
<TensorSliceDataset element_spec=(TensorSpec(shape=(28, 28), dtype=tf.float64, name=None), TensorSpec
```

Note: The above code snippet will embed the **features** and **labels** arrays in your TensorFlow graph as **`tf.constant()`** (https://www.tensorflow.org/api_docs/python/tf/constant) operations. This works well for a small dataset, but wastes memory—because the contents of the array will be copied multiple times—and can run into the 2GB limit for the **`tf.GraphDef`** protocol buffer.

Consuming Python generators

Another common data source that can easily be ingested as a **`tf.data.Dataset`** (https://www.tensorflow.org/api_docs/python/tf/data/Dataset) is the python generator.

Caution: While this is a convenient approach it has limited portability and scalability. It must run in the same python process that created the generator, and is still subject to the Python **`GIL`** (https://en.wikipedia.org/wiki/Global_interpreter_lock).

```
def count(stop):
    i = 0
    while i < stop:
        yield i
        i += 1
```

```
for n in count(5):
    print(n)
```

```
0
1
2
3
4
```

The **`Dataset.from_generator`** (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#from_generator) constructor converts the python generator to a fully functional **`tf.data.Dataset`** (https://www.tensorflow.org/api_docs/python/tf/data/Dataset).

The constructor takes a callable as input, not an iterator. This allows it to restart the generator when it reaches the end. It takes an optional **`args`** argument, which is passed as the callable's arguments.

The **`output_types`** argument is required because **`tf.data`** (https://www.tensorflow.org/api_docs/python/tf/data) builds a **`tf.Graph`** (https://www.tensorflow.org/api_docs/python/tf/Graph) internally, and graph edges require a **`tf.dtype`**.

```
ds_counter = tf.data.Dataset.from_generator(count, args=[25], output_types=tf.int32, output_shapes =
```

```
for count_batch in ds_counter.repeat().batch(10).take(10):
    print(count_batch.numpy())
```

```
[0 1 2 3 4 5 6 7 8 9]
[10 11 12 13 14 15 16 17 18 19]
[20 21 22 23 24 0 1 2 3 4]
[ 5 6 7 8 9 10 11 12 13 14]
[15 16 17 18 19 20 21 22 23 24]
[0 1 2 3 4 5 6 7 8 9]
[10 11 12 13 14 15 16 17 18 19]
[20 21 22 23 24 0 1 2 3 4]
[ 5 6 7 8 9 10 11 12 13 14]
[15 16 17 18 19 20 21 22 23 24]
```

The `output_shapes` argument is not *required* but is highly recommended as many TensorFlow operations do not support tensors with an unknown rank. If the length of a particular axis is unknown or variable, set it as `None` in the `output_shapes`.

It's also important to note that the `output_shapes` and `output_types` follow the same nesting rules as other dataset methods.

Here is an example generator that demonstrates both aspects: it returns tuples of arrays, where the second array is a vector with unknown length.

```
def gen_series():
    i = 0
    while True:
        size = np.random.randint(0, 10)
        yield i, np.random.normal(size=(size,))
        i += 1
```

```
for i, series in gen_series():
    print(i, ":", str(series))
    if i > 5:
        break
```

```
0 : [0.9961]
1 : [-0.1248 -0.3577 -0.6692 -1.3581]
2 : [ 1.9101 -1.0822 -1.2931 -0.2526  0.042  0.2612 -1.5911  0.9461]
3 : [ 1.0877  0.4791 -0.2642  1.2869  0.3199]
4 : [-0.4801 -0.3536  0.0931 -2.5577  0.0535 -0.5872 -1.3122  0.3057 -1.1539]
5 : [-1.1563]
6 : [0.7098 0.8881 0.1062]
```

The first output is an `int32` the second is a `float32`.

The first item is a scalar, shape `()`, and the second is a vector of unknown length, shape `(None,)`

```
ds_series = tf.data.Dataset.from_generator(
    gen_series,
    output_types=(tf.int32, tf.float32),
    output_shapes=([], (None,)))

ds_series
```

```
<FlatMapDataset element_spec=(TensorSpec(shape=(), dtype=tf.int32, name=None), TensorSpec(shape=(None
```

Now it can be used like a regular `tf.data.Dataset` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset). Note that when batching a dataset with a variable shape, you need to use `Dataset.padded_batch` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#padded_batch).

```
ds_series_batch = ds_series.shuffle(20).padded_batch(10)

ids, sequence_batch = next(iter(ds_series_batch))
print(ids.numpy())
print()
print(sequence_batch.numpy())
```

```
[ 8  2  5 22 21 14  6 11  3 13]
```

```
[[ 0.      0.      0.      0.      0.      0.      0.      0.      0.      ]
 [-0.2497 -1.324   1.7601  0.9996 -1.0217  1.4341  0.4693  0.      0.      ]
 [ 1.9685  1.177   0.6144  1.2265 -0.6857  0.5321  1.2144  0.4235 -1.1163]
 [-0.494   -1.2288 -2.285   -0.3774  0.4073  0.9545 -1.6909  0.      0.      ]
 [-0.2521 -0.5219 -0.1872 -1.4575  1.2822 -0.1168  0.5713  1.0132 -0.8849]
 [-0.3716  0.      0.      0.      0.      0.      0.      0.      0.      ]
 [-0.8776  0.      0.      0.      0.      0.      0.      0.      0.      ]
 [-0.4614 -0.0271 -0.8988 -0.1659 -0.5733 -1.7219 -0.1077  0.      0.      ]
 [ 0.34    3.5969  1.2373 -0.5224  1.6537 -0.4134  0.7328  0.      0.      ]
 [ 0.2292 -0.4442  0.1695  0.      0.      0.      0.      0.      0.      ]]
```

For a more realistic example, try wrapping `preprocessing.image.ImageDataGenerator` (https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image/ImageDataGenerator) as a `tf.data.Dataset` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset).

First download the data:

```
flowers = tf.keras.utils.get_file(
    'flower_photos',
    'https://storage.googleapis.com/download.tensorflow.org/example_images/flower_photos.tgz',
    untar=True)
```

```
Downloading data from https://storage.googleapis.com/download.tensorflow.org/example_images/flower_ph
228813984/228813984 [=====] - 1s 0us/step
```

Create the `image.ImageDataGenerator`

(https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image/ImageDataGenerator)

```
img_gen = tf.keras.preprocessing.image.ImageDataGenerator(rescale=1./255, rotation_range=20)
```



```
images, labels = next(img_gen.flow_from_directory(flowers))
```

```
Found 3670 images belonging to 5 classes.
```

```
print(images.dtype, images.shape)
print(labels.dtype, labels.shape)
```

```
float32 (32, 256, 256, 3)
float32 (32, 5)
```

```
ds = tf.data.Dataset.from_generator(
    lambda: img_gen.flow_from_directory(flowers),
    output_types=(tf.float32, tf.float32),
    output_shapes=([32,256,256,3], [32,5])
)

ds.element_spec
```

```
(TensorSpec(shape=(32, 256, 256, 3), dtype=tf.float32, name=None),
 TensorSpec(shape=(32, 5), dtype=tf.float32, name=None))
```

```
for images, labels in ds.take(1):
    print('images.shape: ', images.shape)
    print('labels.shape: ', labels.shape)
```

```
Found 3670 images belonging to 5 classes.
images.shape: (32, 256, 256, 3)
labels.shape: (32, 5)
```

Consuming TFRecord data

Refer to the [Loading TFRecords](https://www.tensorflow.org/tutorials/load_data/tfrecord) (https://www.tensorflow.org/tutorials/load_data/tfrecord) tutorial for an end-to-end example.

The `tf.data` (https://www.tensorflow.org/api_docs/python/tf/data) API supports a variety of file formats so that you can process large datasets that do not fit in memory. For example, the TFRecord file format is a simple record-oriented binary format that many TensorFlow applications use for training data. The `tf.data.TFRecordDataset` (https://www.tensorflow.org/api_docs/python/tf/data/TFRecordDataset) class enables you to stream over the contents of one or more TFRecord files as part of an input pipeline.

Here is an example using the test file from the French Street Name Signs (FSNS).

```
# Creates a dataset that reads all of the examples from two files.
fsns_test_file = tf.keras.utils.get_file("fsns.tfrec", "https://storage.googleapis.com/download.tensorflow.org/data/fsns-20160927/testc7904079/7904079 [=====] - 0s 0us/step
```

Downloading data from <https://storage.googleapis.com/download.tensorflow.org/data/fsns-20160927/testc7904079/7904079> [=====] - 0s 0us/step

The `filenames` argument to the `TFRecordDataset` initializer can either be a string, a list of strings, or a `tf.Tensor` (https://www.tensorflow.org/api_docs/python/tf/Tensor) of strings. Therefore if you have two sets of files for training and validation purposes, you can create a factory method that produces the dataset, taking filenames as an input argument:

```
dataset = tf.data.TFRecordDataset(filenames = [fsns_test_file])
dataset
```

```
<TFRecordDatasetV2 element_spec=TensorSpec(shape=(), dtype=tf.string, name=None)>
```

Many TensorFlow projects use serialized `tf.train.Example` (https://www.tensorflow.org/api_docs/python/tf/train/Example) records in their TFRecord files. These need to be decoded before they can be inspected:

```
raw_example = next(iter(dataset))
parsed = tf.train.Example.FromString(raw_example.numpy())

parsed.features.feature['image/text']
```

```
bytes_list {
  value: "Rue Perreyon"
}
```

Consuming text data

Refer to the [Load text](https://www.tensorflow.org/tutorials/load_data/text) (https://www.tensorflow.org/tutorials/load_data/text) tutorial for an end-to-end example.

Many datasets are distributed as one or more text files. The `tf.data.TextLineDataset` (https://www.tensorflow.org/api_docs/python/tf/data/TextLineDataset) provides an easy way to extract lines from one or more text files. Given one or more filenames, a `TextLineDataset` will produce one string-valued element per line of those files.

```
directory_url = 'https://storage.googleapis.com/download.tensorflow.org/data/illiad/'
file_names = ['cowper.txt', 'derby.txt', 'butler.txt']

file_paths = [
    tf.keras.utils.get_file(file_name, directory_url + file_name)
    for file_name in file_names
]
```

```

Downloading data from https://storage.googleapis.com/download.tensorflow.org/data/illiad/cowper.txt
815980/815980 [=====] - 0s 0us/step
Downloading data from https://storage.googleapis.com/download.tensorflow.org/data/illiad/derby.txt
809730/809730 [=====] - 0s 0us/step
Downloading data from https://storage.googleapis.com/download.tensorflow.org/data/illiad/butler.txt
807992/807992 [=====] - 0s 0us/step

```

```
dataset = tf.data.TextLineDataset(file_paths)
```

Here are the first few lines of the first file:

```

for line in dataset.take(5):
    print(line.numpy())

```

```

b"\xef\xbb\xbfAchilles sing, O Goddess! Peleus' son;"
b'His wrath pernicious, who ten thousand woes'
b"Caused to Achaia's host, sent many a soul"
b'Illustrious into Ades premature,'
b'And Heroes gave (so stood the will of Jove)'

```

To alternate lines between files use `Dataset.interleave`

(https://www.tensorflow.org/api_docs/python/tf/data/Dataset#interleave). This makes it easier to shuffle files together. Here are the first, second and third lines from each translation:

```

files_ds = tf.data.Dataset.from_tensor_slices(file_paths)
lines_ds = files_ds.interleave(tf.data.TextLineDataset, cycle_length=3)

for i, line in enumerate(lines_ds.take(9)):
    if i % 3 == 0:
        print()
    print(line.numpy())

```

```

b"\xef\xbb\xbfAchilles sing, O Goddess! Peleus' son;"
b"\xef\xbb\xbfOf Peleus' son, Achilles, sing, O Muse,"
b'\xef\xbb\xbfSing, O goddess, the anger of Achilles son of Peleus, that brought'

b'His wrath pernicious, who ten thousand woes'
b'The vengeance, deep and deadly; whence to Greece'
b'countless ills upon the Achaeans. Many a brave soul did it send'

b"Caused to Achaia's host, sent many a soul"
b'Unnumbered ills arose; which many a soul'
b'hurrying down to Hades, and many a hero did it yield a prey to dogs and'

```

By default, a `TextLineDataset` yields every line of each file, which may not be desirable, for example, if the file starts with a header line, or contains comments. These lines can be removed using the `Dataset.skip()`

(https://www.tensorflow.org/api_docs/python/tf/data/Dataset#skip) or `Dataset.filter`

(https://www.tensorflow.org/api_docs/python/tf/data/Dataset#filter) transformations. Here, you skip the first line, then filter to find only survivors.

```
titanic_file = tf.keras.utils.get_file("train.csv", "https://storage.googleapis.com/tf-datasets/titanic/train.csv")
titanic_lines = tf.data.TextLineDataset(titanic_file)
```

```
Downloading data from https://storage.googleapis.com/tf-datasets/titanic/train.csv
30874/30874 [=====] - 0s 0us/step
```

```
for line in titanic_lines.take(10):
    print(line.numpy())
```

```
b'survived,sex,age,n_siblings_spouses,parch,fare,class,deck,embark_town,alone'
b'0,male,22.0,1,0,7.25,Third,unknown,Southampton,n'
b'1,female,38.0,1,0,71.2833,First,C,Cherbourg,n'
b'1,female,26.0,0,0,7.925,Third,unknown,Southampton,y'
b'1,female,35.0,1,0,53.1,First,C,Southampton,n'
b'0,male,28.0,0,0,8.4583,Third,unknown,Queenstown,y'
b'0,male,2.0,3,1,21.075,Third,unknown,Southampton,n'
b'1,female,27.0,0,2,11.1333,Third,unknown,Southampton,n'
b'1,female,14.0,1,0,30.0708,Second,unknown,Cherbourg,n'
b'1,female,4.0,1,1,16.7,Third,G,Southampton,n'
```

```
def survived(line):
    return tf.not_equal(tf.strings.substr(line, 0, 1), "0")
```

```
survivors = titanic_lines.skip(1).filter(survived)
```

```
for line in survivors.take(10):
    print(line.numpy())
```

```
b'1,female,38.0,1,0,71.2833,First,C,Cherbourg,n'
b'1,female,26.0,0,0,7.925,Third,unknown,Southampton,y'
b'1,female,35.0,1,0,53.1,First,C,Southampton,n'
b'1,female,27.0,0,2,11.1333,Third,unknown,Southampton,n'
b'1,female,14.0,1,0,30.0708,Second,unknown,Cherbourg,n'
b'1,female,4.0,1,1,16.7,Third,G,Southampton,n'
b'1,male,28.0,0,0,13.0,Second,unknown,Southampton,y'
b'1,female,28.0,0,0,7.225,Third,unknown,Cherbourg,y'
b'1,male,28.0,0,0,35.5,First,A,Southampton,y'
b'1,female,38.0,1,5,31.3875,Third,unknown,Southampton,n'
```

Consuming CSV data

Refer to the [Loading CSV Files](https://www.tensorflow.org/tutorials/load_data/csv) (https://www.tensorflow.org/tutorials/load_data/csv) and [Loading Pandas DataFrames](https://www.tensorflow.org/tutorials/load_data/pandas_dataframe) (https://www.tensorflow.org/tutorials/load_data/pandas_dataframe) tutorials for more examples.

The CSV file format is a popular format for storing tabular data in plain text.

For example:

```
titanic_file = tf.keras.utils.get_file("train.csv", "https://storage.googleapis.com/tf-datasets/titar
```

```
df = pd.read_csv(titanic_file)
df.head()
```

	survived	sex	age	n_siblings_spouses	parch	fare	class	deck	embark_town	alone
0	male	22.0	1	0	7.2500	Third	unknown	Southampton	n	
1	female	38.0	1	0	71.2833	First	C	Cherbourg	n	
1	female	26.0	0	0	7.9250	Third	unknown	Southampton	y	
1	female	35.0	1	0	53.1000	First	C	Southampton	n	
0	male	28.0	0	0	8.4583	Third	unknown	Queenstown	y	

If your data fits in memory the same `Dataset.from_tensor_slices`

(https://www.tensorflow.org/api_docs/python/tf/data/Dataset#from_tensor_slices) method works on dictionaries, allowing this data to be easily imported:

```
titanic_slices = tf.data.Dataset.from_tensor_slices(dict(df))
```

```
for feature_batch in titanic_slices.take(1):
    for key, value in feature_batch.items():
        print("  {:20s}: {}".format(key, value))
```

```
'survived'          : 0
'sex'               : b'male'
'age'               : 22.0
'n_siblings_spouses': 1
'parch'             : 0
'fare'              : 7.25
'class'             : b'Third'
'deck'              : b'unknown'
'embark_town'       : b'Southampton'
'alone'             : b'n'
```

A more scalable approach is to load from disk as necessary.

The `tf.data` (https://www.tensorflow.org/api_docs/python/tf/data) module provides methods to extract records from one or more CSV files that comply with [RFC 4180](https://tools.ietf.org/html/rfc4180) (<https://tools.ietf.org/html/rfc4180>).

The `tf.data.experimental.make_csv_dataset`

(https://www.tensorflow.org/api_docs/python/tf/data/experimental/make_csv_dataset) function is the high-level interface for reading sets of CSV files. It supports column type inference and many other features, like batching and shuffling, to make usage simple.

```
titanic_batches = tf.data.experimental.make_csv_dataset(
    titanic_file, batch_size=4,
    label_name="survived")
```

```
for feature_batch, label_batch in titanic_batches.take(1):
    print("'survived': {}".format(label_batch))
    print("features:")
    for key, value in feature_batch.items():
        print("  {}: {}".format(key, value))
```

```
'survived': [0 0 0 0]
features:
  'sex'          : [b'male' b'male' b'male' b'male']
  'age'          : [16. 20. 40. 28.]
  'n_siblings_spouses': [4 0 1 0]
  'parch'        : [1 0 4 0]
  'fare'         : [39.6875  9.225 27.9   7.725 ]
  'class'        : [b'Third' b'Third' b'Third' b'Third']
  'deck'         : [b'unknown' b'unknown' b'unknown' b'unknown']
  'embark_town'   : [b'Southampton' b'Southampton' b'Southampton' b'Queenstown']
  'alone'        : [b'n' b'y' b'n' b'y']
```

You can use the `select_columns` argument if you only need a subset of columns.

```
titanic_batches = tf.data.experimental.make_csv_dataset(
    titanic_file, batch_size=4,
    label_name="survived", select_columns=['class', 'fare', 'survived'])
```

```
for feature_batch, label_batch in titanic_batches.take(1):
    print("'survived': {}".format(label_batch))
    for key, value in feature_batch.items():
        print("  {}: {}".format(key, value))
```

```
'survived': [0 1 1 0]
  'fare'          : [82.1708  8.6833  7.8542  5.    ]
  'class'         : [b'First' b'Third' b'Third' b'First']
```

There is also a lower-level `experimental.CsvDataset`

(https://www.tensorflow.org/api_docs/python/tf/data/experimental/CsvDataset) class which provides finer grained control. It does not support column type inference. Instead you must specify the type of each column.

```
titanic_types = [tf.int32, tf.string, tf.float32, tf.int32, tf.int32, tf.float32, tf.string, tf.string]
dataset = tf.data.experimental.CsvDataset(titanic_file, titanic_types, header=True)
```

```
for line in dataset.take(10):
    print([item.numpy() for item in line])
```

```
[0, b'male', 22.0, 1, 0, 7.25, b'Third', b'unknown', b'Southampton', b'n']
[1, b'female', 38.0, 1, 0, 71.2833, b'First', b'C', b'Cherbourg', b'n']
[1, b'female', 26.0, 0, 0, 7.925, b'Third', b'unknown', b'Southampton', b'y']
[1, b'female', 35.0, 1, 0, 53.1, b'First', b'C', b'Southampton', b'n']
[0, b'male', 28.0, 0, 0, 8.4583, b'Third', b'unknown', b'Queenstown', b'y']
[0, b'male', 2.0, 3, 1, 21.075, b'Third', b'unknown', b'Southampton', b'n']
[1, b'female', 27.0, 0, 2, 11.1333, b'Third', b'unknown', b'Southampton', b'n']
[1, b'female', 14.0, 1, 0, 30.0708, b'Second', b'unknown', b'Cherbourg', b'n']
[1, b'female', 4.0, 1, 1, 16.7, b'Third', b'G', b'Southampton', b'n']
[0, b'male', 20.0, 0, 0, 8.05, b'Third', b'unknown', b'Southampton', b'y']
```

If some columns are empty, this low-level interface allows you to provide default values instead of column types.

```
%%writefile missing.csv
1,2,3,4
,2,3,4
1,,3,4
1,2,,4
1,2,3,
,,
```

Writing missing.csv

```
# Creates a dataset that reads all of the records from two CSV files, each with
# four float columns which may have missing values.
```

```
record_defaults = [999,999,999,999]
dataset = tf.data.experimental.CsvDataset("missing.csv", record_defaults)
dataset = dataset.map(lambda *items: tf.stack(items))
dataset
```

```
<MapDataset element_spec=TensorSpec(shape=(4,), dtype=tf.int32, name=None)>
```

```
for line in dataset:
    print(line.numpy())
```

```
[1 2 3 4]
[999  2  3  4]
[ 1 999  3  4]
[ 1  2 999  4]
[ 1  2  3 999]
[999 999 999 999]
```

By default, a `CsvDataset` yields every column of every line of the file, which may not be desirable, for example if the file starts with a header line that should be ignored, or if some columns are not required in the input. These lines and fields can be removed with the `header` and `select_cols` arguments respectively.

```
# Creates a dataset that reads all of the records from two CSV files with
# headers, extracting float data from columns 2 and 4.
record_defaults = [999, 999] # Only provide defaults for the selected columns
dataset = tf.data.experimental.CsvDataset("missing.csv", record_defaults, select_cols=[1, 3])
dataset = dataset.map(lambda *items: tf.stack(items))
dataset
```

```
<MapDataset element_spec=TensorSpec(shape=(2,), dtype=tf.int32, name=None)>
```

```
for line in dataset:
    print(line.numpy())
```

```
[2 4]
[2 4]
[999 4]
[2 4]
[ 2 999]
[999 999]
```

Consuming sets of files

There are many datasets distributed as a set of files, where each file is an example.

```
flowers_root = tf.keras.utils.get_file(
    'flower_photos',
    'https://storage.googleapis.com/download.tensorflow.org/example_images/flower_photos.tgz',
    untar=True)
flowers_root = pathlib.Path(flowers_root)
```

Note: these images are licensed CC-BY, see `LICENSE.txt` for details.

The root directory contains a directory for each class:

```
for item in flowers_root.glob("*"):
    print(item.name)
```

```
daisy
dandelion
sunflowers
roses
LICENSE.txt
tulips
```


The files in each class directory are examples:

```
list_ds = tf.data.Dataset.list_files(str(flowers_root/'*/'))

for f in list_ds.take(5):
    print(f.numpy())
```

```
b'/home/kbuilder/.keras/datasets/flower_photos/daisy/172967318_c596d082cc.jpg'
b'/home/kbuilder/.keras/datasets/flower_photos/roses/394990940_7af082cf8d_n.jpg'
b'/home/kbuilder/.keras/datasets/flower_photos/roses/3909587261_f8cd3e7fe7.jpg'
b'/home/kbuilder/.keras/datasets/flower_photos/daisy/4229503616_9b8a42123c_n.jpg'
b'/home/kbuilder/.keras/datasets/flower_photos/roses/4504731519_9a260b6607_n.jpg'
```

Read the data using the `tf.io.read_file` (https://www.tensorflow.org/api_docs/python/tf/io/read_file) function and extract the label from the path, returning `(image, label)` pairs:

```
def process_path(file_path):
    label = tf.strings.split(file_path, os.sep)[-2]
    return tf.io.read_file(file_path), label

labeled_ds = list_ds.map(process_path)
```

```
for image_raw, label_text in labeled_ds.take(1):
    print(repr(image_raw.numpy()[:100]))
    print()
    print(label_text.numpy())
```

```
b'\xff\xd8\xff\xe0\x00\x10JFIF\x00\x01\x01\x00\x00\x01\x00\x01\x00\x00\xff\xfe\x00\x0cAppleMark\n\xff'
b'tulips'
```

Batching dataset elements

Simple batching

The simplest form of batching stacks `n` consecutive elements of a dataset into a single element. The `Dataset.batch()` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#batch) transformation does exactly this, with the same constraints as the `tf.stack()` (https://www.tensorflow.org/api_docs/python/tf/stack) operator, applied to each component of the elements: i.e. for each component *i*, all elements must have a tensor of the exact same shape.

```
inc_dataset = tf.data.Dataset.range(100)
dec_dataset = tf.data.Dataset.range(0, -100, -1)
dataset = tf.data.Dataset.zip((inc_dataset, dec_dataset))
batched_dataset = dataset.batch(4)
```

```
for batch in batched_dataset.take(4):
    print([arr.numpy() for arr in batch])
```

```
[array([0, 1, 2, 3]), array([ 0, -1, -2, -3])]
[array([4, 5, 6, 7]), array([-4, -5, -6, -7])]
[array([ 8,  9, 10, 11]), array([-8, -9, -10, -11])]
[array([12, 13, 14, 15]), array([-12, -13, -14, -15])]
```

While `tf.data` (https://www.tensorflow.org/api_docs/python/tf/data) tries to propagate shape information, the default settings of `Dataset.batch` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#batch) result in an unknown batch size because the last batch may not be full. Note the `Nones` in the shape:

```
batched_dataset
```

```
<BatchDataset element_spec=(TensorSpec(shape=(None,), dtype=tf.int64, name=None), TensorSpec(shape=(None, 4), dtype=tf.int64, name=None))>
```

Use the `drop_remainder` argument to ignore that last batch, and get full shape propagation:

```
batched_dataset = dataset.batch(7, drop_remainder=True)
batched_dataset
```

```
<BatchDataset element_spec=(TensorSpec(shape=(7,), dtype=tf.int64, name=None), TensorSpec(shape=(7, 4), dtype=tf.int64, name=None))>
```

Batching tensors with padding

The above recipe works for tensors that all have the same size. However, many models (including sequence models) work with input data that can have varying size (for example, sequences of different lengths). To handle this case, the `Dataset.padded_batch` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#padded_batch) transformation enables you to batch tensors of different shapes by specifying one or more dimensions in which they may be padded.

```
dataset = tf.data.Dataset.range(100)
dataset = dataset.map(lambda x: tf.fill([tf.cast(x, tf.int32)], x))
dataset = dataset.padded_batch(4, padded_shapes=(None,))
```

```
for batch in dataset.take(2):
    print(batch.numpy())
    print()
```

```
[[0 0 0]
 [1 0 0]
 [2 2 0]
 [3 3 3]]

[[4 4 4 4 0 0 0]
 [5 5 5 5 0 0 0]]
```

```
[6 6 6 6 6 6 0]
[7 7 7 7 7 7 7]]
```

The `Dataset.padded_batch` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#padded_batch) transformation allows you to set different padding for each dimension of each component, and it may be variable-length (signified by `None` in the example above) or constant-length. It is also possible to override the padding value, which defaults to 0.

Training workflows

Processing multiple epochs

The `tf.data` (https://www.tensorflow.org/api_docs/python/tf/data) API offers two main ways to process multiple epochs of the same data.

The simplest way to iterate over a dataset in multiple epochs is to use the `Dataset.repeat()` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#repeat) transformation. First, create a dataset of titanic data:

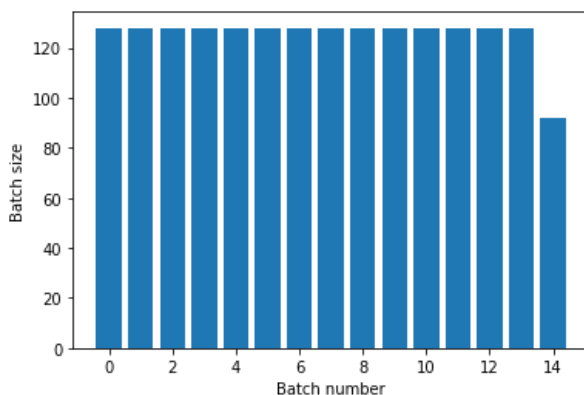
```
titanic_file = tf.keras.utils.get_file("train.csv", "https://storage.googleapis.com/tf-datasets/titanic/train.csv")
titanic_lines = tf.data.TextLineDataset(titanic_file)
```

```
def plot_batch_sizes(ds):
    batch_sizes = [batch.shape[0] for batch in ds]
    plt.bar(range(len(batch_sizes)), batch_sizes)
    plt.xlabel('Batch number')
    plt.ylabel('Batch size')
```

Applying the `Dataset.repeat()` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#repeat) transformation with no arguments will repeat the input indefinitely.

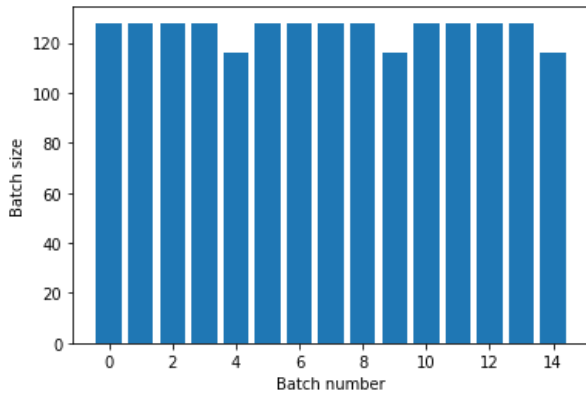
The `Dataset.repeat` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#repeat) transformation concatenates its arguments without signaling the end of one epoch and the beginning of the next epoch. Because of this a `Dataset.batch` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#batch) applied after `Dataset.repeat` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#repeat) will yield batches that straddle epoch boundaries:

```
titanic_batches = titanic_lines.repeat(3).batch(128)
plot_batch_sizes(titanic_batches)
```



If you need clear epoch separation, put `Dataset.batch` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#batch) before the repeat:

```
titanic_batches = titanic_lines.batch(128).repeat(3)
plot_batch_sizes(titanic_batches)
```



If you would like to perform a custom computation (for example, to collect statistics) at the end of each epoch then it's simplest to restart the dataset iteration on each epoch:

```
epochs = 3
dataset = titanic_lines.batch(128)

for epoch in range(epochs):
    for batch in dataset:
        print(batch.shape)
    print("End of epoch: ", epoch)
```

```
(128,)
(128,)
(128,)
(128,)
(116,)
End of epoch:  0
(128,)
(128,)
(128,)
(128,)
(116,)
End of epoch:  1
(128,)
(128,)
```

Randomly shuffling input data

The `Dataset.shuffle()` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#shuffle) transformation maintains a fixed-size buffer and chooses the next element uniformly at random from that buffer.

Note: While large `buffer_sizes` shuffle more thoroughly, they can take a lot of memory, and significant time to fill. Consider using `Dataset.interleave` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#interleave) across files if this becomes a problem.

Add an index to the dataset so you can see the effect:

```
lines = tf.data.TextLineDataset(titanic_file)
counter = tf.data.experimental.Counter()

dataset = tf.data.Dataset.zip((counter, lines))
dataset = dataset.shuffle(buffer_size=100)
dataset = dataset.batch(20)
dataset
```

```
<BatchDataset element_spec=(TensorSpec(shape=(None,), dtype=tf.int64, name=None), TensorSpec(shape=(None, 1), dtype=tf.string, name=None))>
```

Since the `buffer_size` is 100, and the batch size is 20, the first batch contains no elements with an index over 120.

```
n, line_batch = next(iter(dataset))
print(n.numpy())
```

```
[ 94  60  64  68  25  32 104  52  44 108   2  12  41 109 113 102  90  17
 69  43]
```

As with [`Dataset.batch`](https://www.tensorflow.org/api_docs/python/tf/data/Dataset#batch) (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#batch) the order relative to [`Dataset.repeat`](https://www.tensorflow.org/api_docs/python/tf/data/Dataset#repeat) (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#repeat) matters.

[`Dataset.shuffle`](https://www.tensorflow.org/api_docs/python/tf/data/Dataset#shuffle) (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#shuffle) doesn't signal the end of an epoch until the shuffle buffer is empty. So a shuffle placed before a repeat will show every element of one epoch before moving to the next:

```
dataset = tf.data.Dataset.zip((counter, lines))
shuffled = dataset.shuffle(buffer_size=100).batch(10).repeat(2)

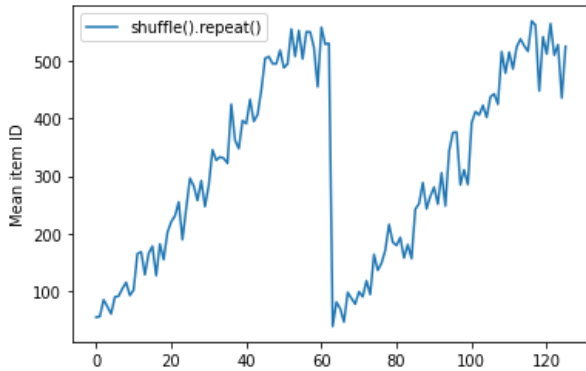
print("Here are the item ID's near the epoch boundary:\n")
for n, line_batch in shuffled.skip(60).take(5):
    print(n.numpy())
```

Here are the item ID's near the epoch boundary:

```
[600 538 607 611 627 594 584 618 411 458]
[579 328 504 536 553 558 620 602 524 608]
[619 443 478 551  89 571 459 302]
[ 10  54  82  66   3 101  14   5  18  69]
[104  97   9  48  42  39  71  12  22 105]
```

```
shuffle_repeat = [n.numpy().mean() for n, line_batch in shuffled]
plt.plot(shuffle_repeat, label="shuffle().repeat()")
plt.ylabel("Mean item ID")
plt.legend()
```

<matplotlib.legend.Legend at 0x7f363fc86ee0>



But a repeat before a shuffle mixes the epoch boundaries together:

```
dataset = tf.data.Dataset.zip((counter, lines))
shuffled = dataset.repeat(2).shuffle(buffer_size=100).batch(10)

print("Here are the item ID's near the epoch boundary:\n")
for n, line_batch in shuffled.skip(55).take(15):
    print(n.numpy())
```

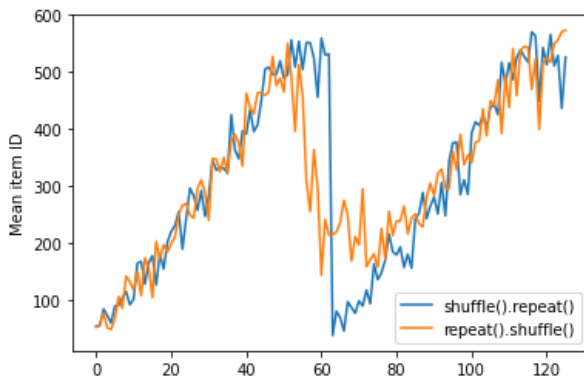
Here are the item ID's near the epoch boundary:

```
[575 494 531 600 588 354 626 491 563 581]
[ 12  20  23 615  35 574  18 544  36 589]
[621 590 496 481  14 618 592 607  27 526]
[604 611 586  47  11 387  30  42  13  32]
[  8 532 465 517 510  58 609  39 469 627]
[598  65 593 623  21  0 421  6  64 513]
[ 16  57  40 450  41  70  63  9  77  44]
[ 34 355  74  94  81  17  1  50 571 580]
[ 80 101 508  26  37 624  33  53 613 107]
[622  59 398 579 100  91  84 489 595 338]
[ 82 113 301 490  95 617 122  28  29 415]
[ 48 131 129  75  52 117 578 118 594 119]
```

```
repeat_shuffle = [n.numpy().mean() for n, line_batch in shuffled]

plt.plot(shuffle_repeat, label="shuffle().repeat()")
plt.plot(repeat_shuffle, label="repeat().shuffle()")
plt.ylabel("Mean item ID")
plt.legend()
```

<matplotlib.legend.Legend at 0x7f363fc42d90>



Preprocessing data

The `Dataset.map(f)` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#map) transformation produces a new dataset by applying a given function `f` to each element of the input dataset. It is based on the `map()` ([https://en.wikipedia.org/wiki/Map_\(higher-order_function\)](https://en.wikipedia.org/wiki/Map_(higher-order_function))) function that is commonly applied to lists (and other structures) in functional programming languages. The function `f` takes the `tf.Tensor` (https://www.tensorflow.org/api_docs/python/tf/Tensor) objects that represent a single element in the input, and returns the `tf.Tensor` (https://www.tensorflow.org/api_docs/python/tf/Tensor) objects that will represent a single element in the new dataset. Its implementation uses standard TensorFlow operations to transform one element into another.

This section covers common examples of how to use `Dataset.map()`. (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#map).

Decoding image data and resizing it

When training a neural network on real-world image data, it is often necessary to convert images of different sizes to a common size, so that they may be batched into a fixed size.

Rebuild the flower filenames dataset:

```
list_ds = tf.data.Dataset.list_files(str(flowers_root/'*/'))
```

Write a function that manipulates the dataset elements.

```
# Reads an image from a file, decodes it into a dense tensor, and resizes it
# to a fixed shape.
def parse_image(filename):
    parts = tf.strings.split(filename, os.sep)
    label = parts[-2]

    image = tf.io.read_file(filename)
    image = tf.io.decode_jpeg(image)
    image = tf.image.convert_image_dtype(image, tf.float32)
    image = tf.image.resize(image, [128, 128])
    return image, label
```

Test that it works.

```
file_path = next(iter(list_ds))
image, label = parse_image(file_path)

def show(image, label):
    plt.figure()
    plt.imshow(image)
    plt.title(label.numpy().decode('utf-8'))
    plt.axis('off')

show(image, label)
```

dandelion

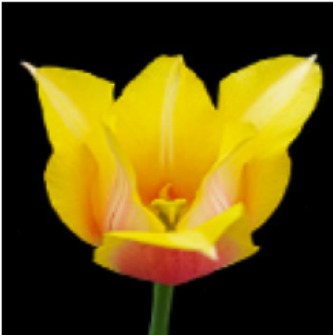


Map it over the dataset.

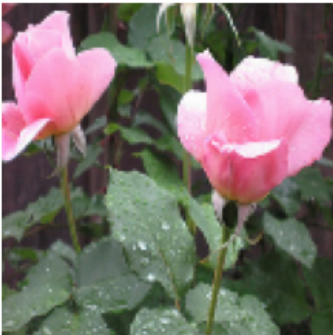
```
images_ds = list_ds.map(parse_image)

for image, label in images_ds.take(2):
    show(image, label)
```

tulips



roses



Applying arbitrary Python logic

For performance reasons, use TensorFlow operations for preprocessing your data whenever possible. However, it is sometimes useful to call external Python libraries when parsing your input data. You can use the `tf.py_function` (https://www.tensorflow.org/api_docs/python/tf/py_function) operation in a `Dataset.map` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#map) transformation.

For example, if you want to apply a random rotation, the `tf.image` (https://www.tensorflow.org/api_docs/python/tf/image) module only has `tf.image.rot90` (https://www.tensorflow.org/api_docs/python/tf/image/rot90), which is not very useful for image augmentation.

Note: `tensorflow-addons` has a TensorFlow compatible `rotate` in `tensorflow-addons.image.rotate`.

To demonstrate `tf.py_function` (https://www.tensorflow.org/api_docs/python/tf/py_function), try using the `scipy.ndimage.rotate` function instead:

```
import scipy.ndimage as ndimage

def random_rotate_image(image):
    image = ndimage.rotate(image, np.random.uniform(-30, 30), reshape=False)
    return image
```

```
image, label = next(iter(images_ds))
image = random_rotate_image(image)
show(image, label)
```

Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers)



To use this function with `Dataset.map` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#map) the same caveats apply as with `Dataset.from_generator` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#from_generator), you need to describe the return shapes and types when you apply the function:

```
def tf_random_rotate_image(image, label):
    im_shape = image.shape
    [image,] = tf.py_function(random_rotate_image, [image], [tf.float32])
    image.set_shape(im_shape)
    return image, label
```

```
rot_ds = images_ds.map(tf_random_rotate_image)

for image, label in rot_ds.take(2):
    show(image, label)
```

Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers)



Parsing `tf.Example` protocol buffer messages

Many input pipelines extract `tf.train.Example` (https://www.tensorflow.org/api_docs/python/tf/train/Example) protocol buffer messages from a TFRecord format. Each `tf.train.Example`

(https://www.tensorflow.org/api_docs/python/tf/train/Example) record contains one or more "features", and the input pipeline typically converts these features into tensors.

```
fsns_test_file = tf.keras.utils.get_file("fsns.tfrec", "https://storage.googleapis.com/download.tensorflow.org/example_data/fsns.tfrec")
dataset = tf.data.TFRecordDataset(filename = [fsns_test_file])
dataset
```

```
<TFRecordDatasetV2 element_spec=TensorSpec(shape=(), dtype=tf.string, name=None)>
```

You can work with `tf.train.Example` (https://www.tensorflow.org/api_docs/python/tf/train/Example) protos outside of a `tf.data.Dataset` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset) to understand the data:

```

raw_example = next(iter(dataset))
parsed = tf.train.Example.FromString(raw_example.numpy())

feature = parsed.features.feature
raw_img = feature['image/encoded'].bytes_list.value[0]
img = tf.image.decode_png(raw_img)
plt.imshow(img)
plt.axis('off')
_ = plt.title(feature["image/text"].bytes_list.value[0])

```

b'Rue Perreyon'



```
raw_example = next(iter(dataset))
```

```

def tf_parse(eg):
    example = tf.io.parse_example(
        eg[tf.newaxis], {
            'image/encoded': tf.io.FixedLenFeature(shape=(), dtype=tf.string),
            'image/text': tf.io.FixedLenFeature(shape=(), dtype=tf.string)
        })
    return example['image/encoded'][0], example['image/text'][0]

```

```

img, txt = tf_parse(raw_example)
print(txt.numpy())
print(repr(img.numpy()[:20]), "...")

```

```

b'Rue Perreyon'
b'\x89PNG\r\n\x1a\n\x00\x00\x00\rIHDR\x00\x00\x02X' ...

```

```

decoded = dataset.map(tf_parse)
decoded

```

```
<MapDataset element_spec=(TensorSpec(shape=(), dtype=tf.string, name=None), TensorSpec(shape=(), dtype=
```

```

image_batch, text_batch = next(iter(decoded.batch(10)))
image_batch.shape

```

```
TensorShape([10])
```

Time series windowing

For an end-to-end time series example see: [Time series forecasting](https://www.tensorflow.org/tutorials/structured_data/time_series) (https://www.tensorflow.org/tutorials/structured_data/time_series).

Time series data is often organized with the time axis intact.

Use a simple `Dataset.range` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#range) to demonstrate:

```
range_ds = tf.data.Dataset.range(100000)
```

Typically, models based on this sort of data will want a contiguous time slice.

The simplest approach would be to batch the data:

Using batch

```
batches = range_ds.batch(10, drop_remainder=True)

for batch in batches.take(5):
    print(batch.numpy())
```

```
[0 1 2 3 4 5 6 7 8 9]
[10 11 12 13 14 15 16 17 18 19]
[20 21 22 23 24 25 26 27 28 29]
[30 31 32 33 34 35 36 37 38 39]
[40 41 42 43 44 45 46 47 48 49]
```

Or to make dense predictions one step into the future, you might shift the features and labels by one step relative to each other:

```
def dense_1_step(batch):
    # Shift features and labels one step relative to each other.
    return batch[:-1], batch[1:]

predict_dense_1_step = batches.map(dense_1_step)

for features, label in predict_dense_1_step.take(3):
    print(features.numpy(), " => ", label.numpy())
```

```
[0 1 2 3 4 5 6 7 8] => [1 2 3 4 5 6 7 8 9]
[10 11 12 13 14 15 16 17 18] => [11 12 13 14 15 16 17 18 19]
[20 21 22 23 24 25 26 27 28] => [21 22 23 24 25 26 27 28 29]
```

To predict a whole window instead of a fixed offset you can split the batches into two parts:

```
batches = range_ds.batch(15, drop_remainder=True)

def label_next_5_steps(batch):
```

```

    return (batch[:-5],    # Inputs: All except the last 5 steps
            batch[-5:])    # Labels: The last 5 steps

predict_5_steps = batches.map(label_next_5_steps)

for features, label in predict_5_steps.take(3):
    print(features.numpy(), " => ", label.numpy())

```

```

[0 1 2 3 4 5 6 7 8 9]  =>  [10 11 12 13 14]
[15 16 17 18 19 20 21 22 23 24]  =>  [25 26 27 28 29]
[30 31 32 33 34 35 36 37 38 39]  =>  [40 41 42 43 44]

```

To allow some overlap between the features of one batch and the labels of another, use [Dataset.zip](https://www.tensorflow.org/api_docs/python/tf/data/Dataset#zip) (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#zip):

```

feature_length = 10
label_length = 3

features = range_ds.batch(feature_length, drop_remainder=True)
labels = range_ds.batch(feature_length).skip(1).map(lambda labels: labels[:label_length])

predicted_steps = tf.data.Dataset.zip((features, labels))

for features, label in predicted_steps.take(5):
    print(features.numpy(), " => ", label.numpy())

```

```

[0 1 2 3 4 5 6 7 8 9]  =>  [10 11 12]
[10 11 12 13 14 15 16 17 18 19]  =>  [20 21 22]
[20 21 22 23 24 25 26 27 28 29]  =>  [30 31 32]
[30 31 32 33 34 35 36 37 38 39]  =>  [40 41 42]
[40 41 42 43 44 45 46 47 48 49]  =>  [50 51 52]

```

Using window

While using [Dataset.batch](https://www.tensorflow.org/api_docs/python/tf/data/Dataset#batch) (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#batch) works, there are situations where you may need finer control. The [Dataset.window](https://www.tensorflow.org/api_docs/python/tf/data/Dataset#window) (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#window) method gives you complete control, but requires some care: it returns a **Dataset of Datasets**. Go to the [Dataset structure](#) (#dataset_structure) section for details.

```

window_size = 5

windows = range_ds.window(window_size, shift=1)
for sub_ds in windows.take(5):
    print(sub_ds)

```

```

<_VariantDataset element_spec=TensorSpec(shape=(), dtype=tf.int64, name=None)>
<_VariantDataset element_spec=TensorSpec(shape=(), dtype=tf.int64, name=None)>
<_VariantDataset element_spec=TensorSpec(shape=(), dtype=tf.int64, name=None)>
<_VariantDataset element_spec=TensorSpec(shape=(), dtype=tf.int64, name=None)>
<_VariantDataset element_spec=TensorSpec(shape=(), dtype=tf.int64, name=None)>

```

The `Dataset.flat_map` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#flat_map) method can take a dataset of datasets and flatten it into a single dataset:

```
for x in windows.flat_map(lambda x: x).take(30):
    print(x.numpy(), end=' ')
```

```
0 1 2 3 4 1 2 3 4 5 2 3 4 5 6 3 4 5 6 7 4 5 6 7 8 5 6 7 8 9
```

In nearly all cases, you will want to `Dataset.batch` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#batch) the dataset first:

```
def sub_to_batch(sub):
    return sub.batch(window_size, drop_remainder=True)

for example in windows.flat_map(sub_to_batch).take(5):
    print(example.numpy())
```

```
[0 1 2 3 4]
[1 2 3 4 5]
[2 3 4 5 6]
[3 4 5 6 7]
[4 5 6 7 8]
```

Now, you can see that the `shift` argument controls how much each window moves over.

Putting this together you might write this function:

```
def make_window_dataset(ds, window_size=5, shift=1, stride=1):
    windows = ds.window(window_size, shift=shift, stride=stride)

    def sub_to_batch(sub):
        return sub.batch(window_size, drop_remainder=True)

    windows = windows.flat_map(sub_to_batch)
    return windows
```

```
ds = make_window_dataset(range_ds, window_size=10, shift = 5, stride=3)
```

```
for example in ds.take(10):
    print(example.numpy())
```

```
[ 0  3  6  9 12 15 18 21 24 27]
[ 5  8 11 14 17 20 23 26 29 32]
[10 13 16 19 22 25 28 31 34 37]
[15 18 21 24 27 30 33 36 39 42]
[20 23 26 29 32 35 38 41 44 47]
[25 28 31 34 37 40 43 46 49 52]
```

```
[30 33 36 39 42 45 48 51 54 57]
[35 38 41 44 47 50 53 56 59 62]
[40 43 46 49 52 55 58 61 64 67]
[45 48 51 54 57 60 63 66 69 72]
```

Then it's easy to extract labels, as before:

```
dense_labels_ds = ds.map(dense_1_step)

for inputs, labels in dense_labels_ds.take(3):
    print(inputs.numpy(), "=>", labels.numpy())
```

```
[ 0  3  6  9 12 15 18 21 24] => [ 3  6  9 12 15 18 21 24 27]
[ 5  8 11 14 17 20 23 26 29] => [ 8 11 14 17 20 23 26 29 32]
[10 13 16 19 22 25 28 31 34] => [13 16 19 22 25 28 31 34 37]
```

Resampling

When working with a dataset that is very class-imbalanced, you may want to resample the dataset. [tf.data](https://www.tensorflow.org/api_docs/python/tf/data) (https://www.tensorflow.org/api_docs/python/tf/data) provides two methods to do this. The credit card fraud dataset is a good example of this sort of problem.

Note: Go to [Classification on imbalanced data](https://www.tensorflow.org/tutorials/structured_data/imbalanced_data) (https://www.tensorflow.org/tutorials/structured_data/imbalanced_data) for a full tutorial.

```
zip_path = tf.keras.utils.get_file(
    origin='https://storage.googleapis.com/download.tensorflow.org/data/creditcard.zip',
    fname='creditcard.zip',
    extract=True)

csv_path = zip_path.replace('.zip', '.csv')
```

```
Downloading data from https://storage.googleapis.com/download.tensorflow.org/data/creditcard.zip
69155632/69155632 [=====] - 1s 0us/step
```

```
creditcard_ds = tf.data.experimental.make_csv_dataset(
    csv_path, batch_size=1024, label_name="Class",
    # Set the column types: 30 floats and an int.
    column_defaults=[float()]*30+[int()])
```

Now, check the distribution of classes, it is highly skewed:

```
def count(counts, batch):
    features, labels = batch
    class_1 = labels == 1
    class_1 = tf.cast(class_1, tf.int32)
```

```
class_0 = labels == 0
class_0 = tf.cast(class_0, tf.int32)

counts['class_0'] += tf.reduce_sum(class_0)
counts['class_1'] += tf.reduce_sum(class_1)

return counts
```

```
counts = creditcard_ds.take(10).reduce(
    initial_state={'class_0': 0, 'class_1': 0},
    reduce_func = count)

counts = np.array([counts['class_0'].numpy(),
                  counts['class_1'].numpy()]).astype(np.float32)

fractions = counts/counts.sum()
print(fractions)
```

```
[0.9964 0.0036]
```

A common approach to training with an imbalanced dataset is to balance it. [tf.data](https://www.tensorflow.org/api_docs/python/tf/data) (https://www.tensorflow.org/api_docs/python/tf/data) includes a few methods which enable this workflow:

Datasets sampling

One approach to resampling a dataset is to use [sample_from_datasets](https://www.tensorflow.org/api_docs/python/tf/data/Dataset#sample_from_datasets). This is more applicable when you have a separate [tf.data.Dataset](https://www.tensorflow.org/api_docs/python/tf/data/Dataset) (https://www.tensorflow.org/api_docs/python/tf/data/Dataset) for each class.

Here, just use filter to generate them from the credit card fraud data:

```
negative_ds = (
    creditcard_ds
    .unbatch()
    .filter(lambda features, label: label==0)
    .repeat())
positive_ds = (
    creditcard_ds
    .unbatch()
    .filter(lambda features, label: label==1)
    .repeat())
```

```
for features, label in positive_ds.batch(10).take(1):
    print(label.numpy())
```

```
[1 1 1 1 1 1 1 1 1 1]
```

To use [tf.data.Dataset.sample_from_datasets](https://www.tensorflow.org/api_docs/python/tf/data/Dataset#sample_from_datasets)

(https://www.tensorflow.org/api_docs/python/tf/data/Dataset#sample_from_datasets) pass the datasets, and the weight for each:


```
balanced_ds = tf.data.Dataset.sample_from_datasets(
    [negative_ds, positive_ds], [0.5, 0.5]).batch(10)
```

Now the dataset produces examples of each class with a 50/50 probability:

```
for features, labels in balanced_ds.take(10):
    print(labels.numpy())
```

```
[1 1 0 1 1 0 0 1 1 0]
[1 1 1 1 1 1 0 0 0 1]
[1 0 1 0 1 1 0 0 1 1]
[0 1 0 1 1 1 0 1 1 0]
[1 1 1 0 0 0 1 1 1 1]
[0 1 1 1 1 1 0 1 1 1]
[1 0 0 1 1 1 0 0 1 0]
[0 1 0 1 0 1 1 0 0 1]
[1 1 1 1 0 1 1 1 1 0]
[0 0 0 0 0 1 0 1 0 0]
```

Rejection resampling

One problem with the above `Dataset.sample_from_datasets`

(https://www.tensorflow.org/api_docs/python/tf/data/Dataset#sample_from_datasets) approach is that it needs a separate

`tf.data.Dataset` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset) per class. You could use `Dataset.filter`

(https://www.tensorflow.org/api_docs/python/tf/data/Dataset#filter) to create those two datasets, but that results in all the data being loaded twice.

The `tf.data.Dataset.rejection_resample`

(https://www.tensorflow.org/api_docs/python/tf/data/Dataset#rejection_resample) method can be applied to a dataset to rebalance it, while only loading it once. Elements will be dropped from the dataset to achieve balance.

The `rejection_resample` method takes a `class_func` argument. This `class_func` is applied to each dataset element, and is used to determine which class an example belongs to for the purposes of balancing.

The goal here is to balance the label distribution, and the elements of `creditcard_ds` are already `(features, label)` pairs. So the `class_func` just needs to return those labels:

```
def class_func(features, label):
    return label
```

The resampling method deals with individual examples, so in this case you must `unbatch` the dataset before applying that method.

The method needs a target distribution, and optionally an initial distribution estimate as inputs.

```
resample_ds = (
    creditcard_ds
    .unbatch()
    .rejection_resample(class_func, target_dist=[0.5, 0.5],
```

```
initial_dist=fractions)

.batch(10))
```

WARNING:tensorflow:From /tmpfs/src/tf_docs_env/lib/python3.9/site-packages/tensorflow/python/data/ops/Instructions for updating:
Use tf.print instead of tf.Print. Note that tf.print returns a no-output operator that directly print

The `rejection_resample` method returns (`class`, `example`) pairs where the `class` is the output of the `class_func`. In this case, the `example` was already a (`feature`, `label`) pair, so use `map` to drop the extra copy of the labels:

```
balanced_ds = resample_ds.map(lambda extra_label, features_and_label: features_and_label)
```

Now the dataset produces examples of each class with a 50/50 probability:

```
for features, labels in balanced_ds.take(10):
    print(labels.numpy())
```

```
Proportion of examples rejected by sampler is high: [0.996386707][0.996386707 0.0036132813][0 1]
Proportion of examples rejected by sampler is high: [0.996386707][0.996386707 0.0036132813][0 1]
Proportion of examples rejected by sampler is high: [0.996386707][0.996386707 0.0036132813][0 1]
Proportion of examples rejected by sampler is high: [0.996386707][0.996386707 0.0036132813][0 1]
Proportion of examples rejected by sampler is high: [0.996386707][0.996386707 0.0036132813][0 1]
Proportion of examples rejected by sampler is high: [0.996386707][0.996386707 0.0036132813][0 1]
Proportion of examples rejected by sampler is high: [0.996386707][0.996386707 0.0036132813][0 1]
Proportion of examples rejected by sampler is high: [0.996386707][0.996386707 0.0036132813][0 1]
Proportion of examples rejected by sampler is high: [0.996386707][0.996386707 0.0036132813][0 1]
Proportion of examples rejected by sampler is high: [0.996386707][0.996386707 0.0036132813][0 1]
[1 1 0 1 1 0 1 0 1 1]
[1 1 0 1 0 1 1 0 0 1]
[1 0 1 0 0 1 1 0 1 0]
[1 0 1 0 1 0 0 0 0 1]
```

Iterator Checkpointing

Tensorflow supports [taking checkpoints](https://www.tensorflow.org/guide/checkpoint) so that when your training process restarts it can restore the latest checkpoint to recover most of its progress. In addition to checkpointing the model variables, you can also checkpoint the progress of the dataset iterator. This could be useful if you have a large dataset and don't want to start the dataset from the beginning on each restart. Note however that iterator checkpoints may be large, since transformations such as [Dataset.shuffle](https://www.tensorflow.org/api_docs/python/tf/data/Dataset#shuffle) and [Dataset.prefetch](https://www.tensorflow.org/api_docs/python/tf/data/Dataset#prefetch) require buffering elements within the iterator.

To include your iterator in a checkpoint, pass the iterator to the `tf.train.Checkpoint` (https://www.tensorflow.org/api_docs/python/tf/train/Checkpoint) constructor.

```
range_ds = tf.data.Dataset.range(20)

iterator = iter(range_ds)
ckpt = tf.train.Checkpoint(step=tf.Variable(0), iterator=iterator)
```

```

manager = tf.train.CheckpointManager(ckpt, '/tmp/my_ckpt', max_to_keep=3)

print([next(iterator).numpy() for _ in range(5)])

save_path = manager.save()

print([next(iterator).numpy() for _ in range(5)])

ckpt.restore(manager.latest_checkpoint)

print([next(iterator).numpy() for _ in range(5)])

```

```

[0, 1, 2, 3, 4]
[5, 6, 7, 8, 9]
[5, 6, 7, 8, 9]

```

Note: It is not possible to checkpoint an iterator which relies on an external state, such as a [tf.py_function](https://www.tensorflow.org/api_docs/python/tf/py_function) (https://www.tensorflow.org/api_docs/python/tf/py_function). Attempting to do so will raise an exception complaining about the external state.

Using [tf.data](https://www.tensorflow.org/api_docs/python/tf/data) (https://www.tensorflow.org/api_docs/python/tf/data) with [tf.keras](https://www.tensorflow.org/api_docs/python/tf/keras)

(https://www.tensorflow.org/api_docs/python/tf/keras)

The [tf.keras](https://www.tensorflow.org/api_docs/python/tf/keras) (https://www.tensorflow.org/api_docs/python/tf/keras) API simplifies many aspects of creating and executing machine learning models. Its [Model.fit](https://www.tensorflow.org/api_docs/python/tf/keras/Model#fit) (https://www.tensorflow.org/api_docs/python/tf/keras/Model#fit) and [Model.evaluate](https://www.tensorflow.org/api_docs/python/tf/keras/Model#evaluate) (https://www.tensorflow.org/api_docs/python/tf/keras/Model#evaluate) and [Model.predict](https://www.tensorflow.org/api_docs/python/tf/keras/Model#predict) (https://www.tensorflow.org/api_docs/python/tf/keras/Model#predict) APIs support datasets as inputs. Here is a quick dataset and model setup:

```

train, test = tf.keras.datasets.fashion_mnist.load_data()

images, labels = train
images = images/255.0
labels = labels.astype(np.int32)

fmnist_train_ds = tf.data.Dataset.from_tensor_slices((images, labels))
fmnist_train_ds = fmnist_train_ds.shuffle(5000).batch(32)

model = tf.keras.Sequential([
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(10)
])

model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

```

Passing a dataset of (feature, label) pairs is all that's needed for [Model.fit](https://www.tensorflow.org/api_docs/python/tf/keras/Model#fit) (https://www.tensorflow.org/api_docs/python/tf/keras/Model#fit) and [Model.evaluate](https://www.tensorflow.org/api_docs/python/tf/keras/Model#evaluate)

(https://www.tensorflow.org/api_docs/python/tf/keras/Model#evaluate):

```
model.fit(fmnist_train_ds, epochs=2)
```

```
Epoch 1/2
1875/1875 [=====] - 5s 2ms/step - loss: 0.6020 - accuracy: 0.7942
Epoch 2/2
1875/1875 [=====] - 3s 2ms/step - loss: 0.4623 - accuracy: 0.8403
<keras.callbacks.History at 0x7f36b7393400>
```

If you pass an infinite dataset, for example by calling `Dataset.repeat`

(https://www.tensorflow.org/api_docs/python/tf/data/Dataset#repeat), you just need to also pass the `steps_per_epoch` argument:

```
model.fit(fmnist_train_ds.repeat(), epochs=2, steps_per_epoch=20)
```

```
Epoch 1/2
20/20 [=====] - 0s 2ms/step - loss: 0.4028 - accuracy: 0.8562
Epoch 2/2
20/20 [=====] - 0s 2ms/step - loss: 0.4589 - accuracy: 0.8391
<keras.callbacks.History at 0x7f36081553d0>
```

For evaluation you can pass the number of evaluation steps:

```
loss, accuracy = model.evaluate(fmnist_train_ds)
print("Loss :", loss)
print("Accuracy :", accuracy)
```

```
1875/1875 [=====] - 3s 2ms/step - loss: 0.4471 - accuracy: 0.8438
Loss : 0.447142630815506
Accuracy : 0.8438166379928589
```

For long datasets, set the number of steps to evaluate:

```
loss, accuracy = model.evaluate(fmnist_train_ds.repeat(), steps=10)
print("Loss :", loss)
print("Accuracy :", accuracy)
```

```
10/10 [=====] - 0s 2ms/step - loss: 0.4051 - accuracy: 0.8375
Loss : 0.40512052178382874
Accuracy : 0.8374999761581421
```

The labels are not required when calling `Model.predict` (https://www.tensorflow.org/api_docs/python/tf/keras/Model#predict)

.

```
predict_ds = tf.data.Dataset.from_tensor_slices(images).batch(32)
result = model.predict(predict_ds, steps = 10)
print(result.shape)
```

```
10/10 [=====] - 0s 1ms/step
(320, 10)
```

But the labels are ignored if you do pass a dataset containing them:

```
result = model.predict(fmnist_train_ds, steps = 10)
print(result.shape)
```

```
10/10 [=====] - 0s 1ms/step
(320, 10)
```

Except as otherwise noted, the content of this page is licensed under the [Creative Commons Attribution 4.0 License](https://creativecommons.org/licenses/by/4.0/) (<https://creativecommons.org/licenses/by/4.0/>), and code samples are licensed under the [Apache 2.0 License](https://www.apache.org/licenses/LICENSE-2.0) (<https://www.apache.org/licenses/LICENSE-2.0>). For details, see the [Google Developers Site Policies](https://developers.google.com/site-policies) (<https://developers.google.com/site-policies>). Java is a registered trademark of Oracle and/or its affiliates.

Last updated 2022-06-09 UTC.