TFRecord and tf.Example

Learning Objectives

1. Understand the TFRecord format for storing data
2. Understand the tf.Example message type
3. Read and Write a TFRecord file

Introduction

In this notebook, you create, parse, and use the `tf.Example` message, and then serialize, write, and read `tf.Example` messages to and from `.tfrecord` files. To read data efficiently it can be helpful to serialize your data and store it in a set of files (100-200MB each) that can each be read linearly. This is especially true if the data is being streamed over a network. This can also be useful for caching any data-preprocessing.

Each learning objective will correspond to a **#TODO** in the student lab notebook -- try to complete that notebook first before reviewing this solution notebook.

The TFRecord format

The TFRecord format is a simple format for storing a sequence of binary records. Protocol buffers are a cross-platform, cross-language library for efficient serialization of structured data. Protocol messages are defined by `.proto` files, these are often the easiest way to understand a message type.

The `tf.Example` message (or protobuf) is a flexible message type that represents a `{"string": value}` mapping. It is designed for use with TensorFlow and is used throughout the higher-level APIs such as TFX. Note: While useful, these structures are optional. There is no need to convert existing code to use TFRecords, unless you are using `tf.data` and reading data is still the bottleneck to training. See Data Input Pipeline Performance for dataset performance tips.

Load necessary libraries

We will start by importing the necessary libraries for this lab.

In [1]:

```
# Run the chown command to change the ownership of the repository
!sudo chown -R jupyter:jupyter /home/jupyter/training-data-analyst
```

In [2]:

```
# You can use any Python source file as a module by executing an import statement in so
# The import statement combines two operations; it searches for the named module, then
# to a name in the local scope.
!pip install -q tf-nightly
import tensorflow as tf
```

```
import numpy as np
import IPython.display as display

print("TensorFlow version: ",tf.version.VERSION)
```

```
ERROR: pip's dependency resolver does not currently take into account all the packages t
hat are installed. This behaviour is the source of the following dependency conflicts.
tfx 0.28.0 requires attrs<21,>=19.3.0, but you have attrs 21.2.0 which is incompatible.
tfx 0.28.0 requires docker<5,>=4.1, but you have docker 5.0.0 which is incompatible.
tfx 0.28.0 requires google-api-python-client<2,>=1.7.8, but you have google-api-python-c
lient 2.3.0 which is incompatible.
tfx 0.28.0 requires kubernetes<12,>=10.0.1, but you have kubernetes 12.0.1 which is inco
mpatible.
tfx 0.28.0 requires pyarrow<3,>=1, but you have pyarrow 4.0.0 which is incompatible.
tensorflow 2.4.1 requires gast==0.3.3, but you have gast 0.4.0 which is incompatible.
tensorflow 2.4.1 requires grpcio~=1.32.0, but you have grpcio 1.37.1 which is incompatib
le.
tensorflow 2.4.1 requires h5py~=2.10.0, but you have h5py 3.1.0 which is incompatible.
tensorflow-transform 0.28.0 requires pyarrow<3,>=1, but you have pyarrow 4.0.0 which is
 incompatible.
tensorflow-probability 0.11.0 requires cloudpickle==1.3, but you have cloudpickle 1.6.0
 which is incompatible.
tensorflow-model-analysis 0.28.0 requires pyarrow<3,>=1, but you have pyarrow 4.0.0 whic
h is incompatible.
tensorflow-data-validation 0.28.0 requires joblib<0.15,>=0.12, but you have joblib 1.0.1
which is incompatible.
tensorflow-data-validation 0.28.0 requires pyarrow<3,>=1, but you have pyarrow 4.0.0 whi
ch is incompatible.
ml-pipelines-sdk 0.28.0 requires docker<5,>=4.1, but you have docker 5.0.0 which is inco
mpatible.
ml-metadata 0.28.0 requires attrs<21,>=20.3, but you have attrs 21.2.0 which is incompat
ible.
jupyterlab-git 0.11.0 requires nbdime<2.0.0,>=1.1.0, but you have nbdime 3.0.0 which is
 incompatible.
cloud-tpu-client 0.10 requires google-api-python-client==1.8.0, but you have google-api-
python-client 2.3.0 which is incompatible.
apache-beam 2.28.0 requires httplib2<0.18.0,>=0.8, but you have httplib2 0.19.1 which is
incompatible.
apache-beam 2.28.0 requires pyarrow<3.0.0,>=0.15.1, but you have pyarrow 4.0.0 which is
 incompatible.
TensorFlow version:  2.6.0-dev20210520
```

Please ignore any incompatibility warnings and errors.

`tf.Example`

Data types for `tf.Example`

Fundamentally, a `tf.Example` is a `{"string": tf.train.Feature}` mapping.

The `tf.train.Feature` message type can accept one of the following three types (See the `.proto` file for reference). Most other generic types can be coerced into one of these:

1. `tf.train.BytesList` (the following types can be coerced)
   - `string`
   - `byte`
2. `tf.train.FloatList` (the following types can be coerced)

- `float` (`float32`)
- `double` (`float64`)

3. `tf.train.Int64List` (the following types can be coerced)

- `bool`
- `enum`
- `int32`
- `uint32`
- `int64`
- `uint64`

In order to convert a standard TensorFlow type to a `tf.Example` -compatible `tf.train.Feature` , you can use the shortcut functions below. Note that each function takes a scalar input value and returns a `tf.train.Feature` containing one of the three `list` types above:

In [3]:

```
# TODO 1a
# The following functions can be used to convert a value to a type compatible
# with tf.Example.

def _bytes_feature(value):
  """Returns a bytes_list from a string / byte."""
  if isinstance(value, type(tf.constant(0))):
    value = value.numpy() # BytesList won't unpack a string from an EagerTensor.
  return tf.train.Feature(bytes_list=tf.train.BytesList(value=[value]))

def _float_feature(value):
  """Returns a float_list from a float / double."""
  return tf.train.Feature(float_list=tf.train.FloatList(value=[value]))

def _int64_feature(value):
  """Returns an int64_list from a bool / enum / int / uint."""
  return tf.train.Feature(int64_list=tf.train.Int64List(value=[value]))
```

Note: To stay simple, this example only uses scalar inputs. The simplest way to handle non-scalar features is to use `tf.serialize_tensor` to convert tensors to binary-strings. Strings are scalars in tensorflow. Use `tf.parse_tensor` to convert the binary-string back to a tensor.

Below are some examples of how these functions work. Note the varying input types and the standardized output types. If the input type for a function does not match one of the coercible types stated above, the function will raise an exception (e.g. `_int64_feature(1.0)` will error out, since `1.0` is a float, so should be used with the `_float_feature` function instead):

In [4]:

```
print(_bytes_feature(b'test_string'))
print(_bytes_feature(u'test_bytes'.encode('utf-8')))

print(_float_feature(np.exp(1)))

print(_int64_feature(True))
print(_int64_feature(1))
```

```
bytes_list {
  value: "test_string"
```

```
}

bytes_list {
  value: "test_bytes"
}

float_list {
  value: 2.7182817459106445
}

int64_list {
  value: 1
}

int64_list {
  value: 1
}
```

All proto messages can be serialized to a binary-string using the `.SerializeToString` method:

In [5]:

```
# TODO 1b
feature = _float_feature(np.exp(1))

# `SerializeToString()` serializes the message and returns it as a string
feature.SerializeToString()
```

Out[5]:

```
b'\x12\x06\n\x04T\xf8-@'
```

Creating a `tf.Example` message

Suppose you want to create a `tf.Example` message from existing data. In practice, the dataset may come from anywhere, but the procedure of creating the `tf.Example` message from a single observation will be the same:

1. Within each observation, each value needs to be converted to a `tf.train.Feature` containing one of the 3 compatible types, using one of the functions above.
2. You create a map (dictionary) from the feature name string to the encoded feature value produced in #1.
3. The map produced in step 2 is converted to a `Features` message.

In this notebook, you will create a dataset using NumPy.

This dataset will have 4 features:

- a boolean feature, `False` or `True` with equal probability
- an integer feature uniformly randomly chosen from `[0, 5]`
- a string feature generated from a string table by using the integer feature as an index
- a float feature from a standard normal distribution

Consider a sample consisting of 10,000 independently and identically distributed observations from each of the above distributions:

In [ ]:
```
# The number of observations in the dataset.
n_observations = int(1e4)

# Boolean feature, encoded as False or True.
feature0 = np.random.choice([False, True], n_observations)

# Integer feature, random from 0 to 4.
feature1 = np.random.randint(0, 5, n_observations)

# String feature
strings = np.array([b'cat', b'dog', b'chicken', b'horse', b'goat'])
feature2 = strings[feature1]

# Float feature, from a standard normal distribution
feature3 = np.random.randn(n_observations)
```

Each of these features can be coerced into a `tf.Example` -compatible type using one of `_bytes_feature` , `_float_feature` , `_int64_feature` . You can then create a `tf.Example` message from these encoded features:

In [ ]:
```
def serialize_example(feature0, feature1, feature2, feature3):
  """
  Creates a tf.Example message ready to be written to a file.
  """
  # Create a dictionary mapping the feature name to the tf.Example-compatible
  # data type.
  feature = {
      'feature0': _int64_feature(feature0),
      'feature1': _int64_feature(feature1),
      'feature2': _bytes_feature(feature2),
      'feature3': _float_feature(feature3),
  }

  # Create a Features message using tf.train.Example.

  example_proto = tf.train.Example(features=tf.train.Features(feature=feature))
  return example_proto.SerializeToString()
```

For example, suppose you have a single observation from the dataset, `[False, 4, bytes('goat'), 0.9876]` . You can create and print the `tf.Example` message for this observation using `create_message()` . Each single observation will be written as a `Features` message as per the above. Note that the `tf.Example` message is just a wrapper around the `Features` message:

In [ ]:
```
# This is an example observation from the dataset.

example_observation = []

serialized_example = serialize_example(False, 4, b'goat', 0.9876)
serialized_example
```

You can parse TFRecords using the standard protocol buffer `.FromString` method

To decode the message use the `tf.train.Example.FromString` method.

In [ ]:
```
# TODO 1c
example_proto = tf.train.Example.FromString(serialized_example)
example_proto
```

TFRecords format details

A TFRecord file contains a sequence of records. The file can only be read sequentially.

Each record contains a byte-string, for the data-payload, plus the data-length, and CRC32C (32-bit CRC using the Castagnoli polynomial) hashes for integrity checking.

Each record is stored in the following formats:

```
uint64 length
uint32 masked_crc32_of_length
byte   data[length]
uint32 masked_crc32_of_data
```

The records are concatenated together to produce the file. CRCs are described here, and the mask of a CRC is:

```
masked_crc = ((crc >> 15) | (crc << 17)) + 0xa282ead8ul
```

Note: There is no requirement to use `tf.Example` in TFRecord files. `tf.Example` is just a method of serializing dictionaries to byte-strings. Lines of text, encoded image data, or serialized tensors (using `tf.io.serialize_tensor`, and `tf.io.parse_tensor` when loading). See the `tf.io` module for more options.

TFRecord files using `tf.data`

The `tf.data` module also provides tools for reading and writing data in TensorFlow.

Writing a TFRecord file

The easiest way to get the data into a dataset is to use the `from_tensor_slices` method.

Applied to an array, it returns a dataset of scalars:

In [ ]:
```
tf.data.Dataset.from_tensor_slices(feature1)
```

Applied to a tuple of arrays, it returns a dataset of tuples:

In [ ]:
```
features_dataset = tf.data.Dataset.from_tensor_slices((feature0, feature1, feature2, fe
features_dataset
```

In [ ]:

```
# Use `take(1)` to only pull one example from the dataset.
for f0,f1,f2,f3 in features_dataset.take(1):
  print(f0)
  print(f1)
  print(f2)
  print(f3)
```

Use the `tf.data.Dataset.map` method to apply a function to each element of a `Dataset`.

The mapped function must operate in TensorFlow graph mode—it must operate on and return `tf.Tensors`. A non-tensor function, like `serialize_example`, can be wrapped with `tf.py_function` to make it compatible.

Using `tf.py_function` requires to specify the shape and type information that is otherwise unavailable:

In [ ]:

```
 # TODO 2a
def tf_serialize_example(f0,f1,f2,f3):
  tf_string = tf.py_function(
    serialize_example,
    (f0,f1,f2,f3),  # pass these args to the above function.
    tf.string)      # the return type is `tf.string`.
  return tf.reshape(tf_string, ()) # The result is a scalar
```

In [ ]:

```
tf_serialize_example(f0,f1,f2,f3)
```

Apply this function to each element in the dataset:

In [ ]:

```
 # TODO 2b
# `.map` function maps across the elements of the dataset.
serialized_features_dataset = features_dataset.map(tf_serialize_example)
serialized_features_dataset
```

In [ ]:

```
def generator():
  for features in features_dataset:
    yield serialize_example(*features)
```

In [ ]:

```
# Create a Dataset whose elements are generated by generator using `.from_generator` fu
serialized_features_dataset = tf.data.Dataset.from_generator(
    generator, output_types=tf.string, output_shapes=())
```

In [ ]:

```
serialized_features_dataset
```

And write them to a TFRecord file:

In [ ]:

```
filename = 'test.tfrecord'
# `.TFRecordWriter` function writes a dataset to a TFRecord file
```

```
writer = tf.data.experimental.TFRecordWriter(filename)
writer.write(serialized_features_dataset)
```

Reading a TFRecord file

You can also read the TFRecord file using the `tf.data.TFRecordDataset` class.

More information on consuming TFRecord files using `tf.data` can be found here.

Using `TFRecordDataset` s can be useful for standardizing input data and optimizing performance.

In [ ]:
```
 # TODO 2c
filenames = [filename]
raw_dataset = tf.data.TFRecordDataset(filenames)
raw_dataset
```

At this point the dataset contains serialized `tf.train.Example` messages. When iterated over it returns these as scalar string tensors.

Use the `.take` method to only show the first 10 records.

Note: iterating over a `tf.data.Dataset` only works with eager execution enabled.

In [ ]:
```
# Use the `.take` method to pull ten examples from the dataset.
for raw_record in raw_dataset.take(10):
  print(repr(raw_record))
```

These tensors can be parsed using the function below. Note that the `feature_description` is necessary here because datasets use graph-execution, and need this description to build their shape and type signature:

In [ ]:
```
# Create a description of the features.
feature_description = {
    'feature0': tf.io.FixedLenFeature([], tf.int64, default_value=0),
    'feature1': tf.io.FixedLenFeature([], tf.int64, default_value=0),
    'feature2': tf.io.FixedLenFeature([], tf.string, default_value=''),
    'feature3': tf.io.FixedLenFeature([], tf.float32, default_value=0.0),
}

def _parse_function(example_proto):
  # Parse the input `tf.Example` proto using the dictionary above.
  return tf.io.parse_single_example(example_proto, feature_description)
```

Alternatively, use `tf.parse example` to parse the whole batch at once. Apply this function to each item in the dataset using the `tf.data.Dataset.map` method:

In [ ]:
```
parsed_dataset = raw_dataset.map(_parse_function)
parsed_dataset
```

Use eager execution to display the observations in the dataset. There are 10,000 observations in this dataset, but you will only display the first 10. The data is displayed as a dictionary of features. Each item is a `tf.Tensor` , and the `numpy` element of this tensor displays the value of the feature:

In [ ]:
```
for parsed_record in parsed_dataset.take(10):
  print(repr(parsed_record))
```

Here, the `tf.parse_example` function unpacks the `tf.Example` fields into standard tensors.

TFRecord files in Python

The `tf.io` module also contains pure-Python functions for reading and writing TFRecord files.

Writing a TFRecord file

Next, write the 10,000 observations to the file `test.tfrecord`. Each observation is converted to a `tf.Example` message, then written to file. You can then verify that the file `test.tfrecord` has been created:

In [ ]:
```
# Write the `tf.Example` observations to the file.
with tf.io.TFRecordWriter(filename) as writer:
  for i in range(n_observations):
    example = serialize_example(feature0[i], feature1[i], feature2[i], feature3[i])
    writer.write(example)
```

In [ ]:
```
# `du` stands for disk usage and is used to estimate the amount of disk space used by a
!du -sh {filename}
```

Reading a TFRecord file

These serialized tensors can be easily parsed using `tf.train.Example.ParseFromString`:

In [ ]:
```
filenames = [filename]
raw_dataset = tf.data.TFRecordDataset(filenames)
raw_dataset
```

In [ ]:
```
for raw_record in raw_dataset.take(1):
  example = tf.train.Example()
  example.ParseFromString(raw_record.numpy())
  print(example)
```

Walkthrough: Reading and writing image data

This is an end-to-end example of how to read and write image data using TFRecords. Using an image as input data, you will write the data as a TFRecord file, then read the file back and display the image.

This can be useful if, for example, you want to use several models on the same input dataset. Instead of storing the image data raw, it can be preprocessed into the TFRecords format, and that can be used in all further processing and modelling.

First, let's download this image of a cat in the snow and this photo of the Williamsburg Bridge, NYC under construction.

Fetch the images

In [ ]:

```
# Downloads a file from a URL if it not already in the cache using `tf.keras.utils.get_
cat_in_snow  = tf.keras.utils.get_file('320px-Felis_catus-cat_on_snow.jpg', 'https://st
williamsburg_bridge = tf.keras.utils.get_file('194px-New_East_River_Bridge_from_Brookly
```

In [ ]:

```
# Check the image file
display.display(display.Image(filename=cat_in_snow))
display.display(display.HTML('Image cc-by: <a "href=https://commons.wikimedia.org/wiki/
```

In [ ]:

```
display.display(display.Image(filename=williamsburg_bridge))
display.display(display.HTML('<a "href=https://commons.wikimedia.org/wiki/File:New_East
```

Write the TFRecord file

As before, encode the features as types compatible with `tf.Example` . This stores the raw image string feature, as well as the height, width, depth, and arbitrary `label` feature. The latter is used when you write the file to distinguish between the cat image and the bridge image. Use `0` for the cat image, and `1` for the bridge image:

In [ ]:

```
image_labels = {
    cat_in_snow : 0,
    williamsburg_bridge : 1,
}
```

In [ ]:

```
# This is an example, just using the cat image.
image_string = open(cat_in_snow, 'rb').read()

label = image_labels[cat_in_snow]

# Create a dictionary with features that may be relevant.
def image_example(image_string, label):
  image_shape = tf.image.decode_jpeg(image_string).shape

  feature = {
      'height': _int64_feature(image_shape[0]),
      'width': _int64_feature(image_shape[1]),
      'depth': _int64_feature(image_shape[2]),
      'label': _int64_feature(label),
      'image_raw': _bytes_feature(image_string),
  }

  return tf.train.Example(features=tf.train.Features(feature=feature))

for line in str(image_example(image_string, label)).split('\n')[:15]:
  print(line)
print('...')
```

Notice that all of the features are now stored in the `tf.Example` message. Next, functionalize the code above and write the example messages to a file named `images.tfrecords` :

In [ ]:
```
# Write the raw image files to `images.tfrecords`.
# First, process the two images into `tf.Example` messages.
# Then, write to a `.tfrecords` file.
record_file = 'images.tfrecords'
with tf.io.TFRecordWriter(record_file) as writer:
  for filename, label in image_labels.items():
    image_string = open(filename, 'rb').read()
    tf_example = image_example(image_string, label)
    writer.write(tf_example.SerializeToString())
```

In [ ]:
```
# `du` stands for disk usage and is used to estimate the amount of disk space used by a
!du -sh {record_file}
```

Read the TFRecord file

You now have the file— `images.tfrecords` —and can now iterate over the records in it to read back what you wrote. Given that in this example you will only reproduce the image, the only feature you will need is the raw image string. Extract it using the getters described above, namely `example.features.feature['image_raw'].bytes_list.value[0]` . You can also use the labels to determine which record is the cat and which one is the bridge:

In [ ]:
```
raw_image_dataset = tf.data.TFRecordDataset('images.tfrecords')

# Create a dictionary describing the features.
image_feature_description = {
    'height': tf.io.FixedLenFeature([], tf.int64),
    'width': tf.io.FixedLenFeature([], tf.int64),
    'depth': tf.io.FixedLenFeature([], tf.int64),
    'label': tf.io.FixedLenFeature([], tf.int64),
    'image_raw': tf.io.FixedLenFeature([], tf.string),
}

def _parse_image_function(example_proto):
  # Parse the input tf.Example proto using the dictionary above.
  return tf.io.parse_single_example(example_proto, image_feature_description)

parsed_image_dataset = raw_image_dataset.map(_parse_image_function)
parsed_image_dataset
```

Recover the images from the TFRecord file:

In [ ]:
```
for image_features in parsed_image_dataset:
  image_raw = image_features['image_raw'].numpy()
  display.display(display.Image(data=image_raw))
```