Using the `Dataset` API for TensorFlow Input Pipelines

The `Dataset` 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 `Dataset` API makes it easy to deal with large amounts of data, different data formats, and complicated transformations.

The `Dataset` API introduces two new abstractions to TensorFlow:

- A `tf.contrib.data.Dataset` represents a sequence of elements, in which each element contains one or more `Tensor` objects. For example, in an image pipeline, an element might be a single training example, with a pair of tensors representing the image data and a label. There are two distinct ways to create a dataset:
- Creating a **source** (e.g. `Dataset.from_tensor_slices()`) constructs a dataset from one or more `tf.Tensor` objects.
- Applying a **transformation** (e.g. `Dataset.batch()`) constructs a dataset from one or more `tf.contrib.data.Dataset` objects.
- A `tf.contrib.data.Iterator` provides the main way to extract elements from a dataset. The operation returned by `Iterator.get_next()` yields the next element of a `Dataset` when executed, and typically acts as the interface between input pipeline code and your model. The simplest iterator is a "one-shot iterator", which is associated with a particular `Dataset` and iterates through it once. For more sophisticated uses, the `Iterator.initializer` operation enables you to reinitialize and parameterize an iterator with different datasets, so that you can, for example, iterate over training and validation data multiple times in the same program.

Basic mechanics

This section of the guide describes the fundamentals of creating different kinds of `Dataset` and `Iterator` objects, and how to extract data from them.

To start an input pipeline, you must define a *source*. For example, to construct a `Dataset` from some tensors in memory, you can use `tf.contrib.data.Dataset.from_tensors()`

or `tf.contrib.data.Dataset.from_tensor_slices()`. Alternatively, if your input data are on disk in the recommend TFRecord format, you can construct a `tf.contrib.data.TFRecordDataset`.

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

The most common way to consume values from a `Dataset` is to make an **iterator** object that provides access to one element of the dataset at a time (for example, by calling `Dataset.make_one_shot_iterator()`). A `tf.contrib.data.Iterator` provides two operations: `Iterator.initializer`, which enables you to (re)initialize the iterator's state; and `Iterator.get_next()`, which returns `tf.Tensor` objects that correspond to the symbolic next element. Depending on your use case, you might choose a different type of iterator, and the options are outlined below.

Dataset structure

A dataset comprises elements that each have the same structure. An element contains one or more `tf.Tensor` objects, called *components*. Each component has a `tf.DType` representing the type of elements in the tensor, and a `tf.TensorShape` representing the (possibly partially specified) static shape of each element. The `Dataset.output_types` and `Dataset.output_shapes` properties allow you to inspect the inferred types and shapes of each component of a dataset element. The *nested structure* of these properties map to the structure of an element, which may be a single tensor, a tuple of tensors, or a nested tuple of tensors. For example:

```
dataset1 = tf.contrib.data.Dataset.from_tensor_slices(tf.random_uniform([4, 1
print(dataset1.output_types)  # ==> "tf.float32"
print(dataset1.output_shapes)  # ==> "(10,)"

dataset2 = tf.contrib.data.Dataset.from_tensor_slices(
   (tf.random_uniform([4]),
    tf.random_uniform([4, 100], maxval=100, dtype=tf.int32)))
print(dataset2.output_types)  # ==> "(tf.float32, tf.int32)"
print(dataset2.output_shapes)  # ==> "((), (100,))"

dataset3 = tf.contrib.data.Dataset.zip((dataset1, dataset2))
print(dataset3.output_types)  # ==> (tf.float32, (tf.float32, tf.int32))
print(dataset3.output_shapes)  # ==> "(10, ((), (100,)))"
```

It is often convenient to give names to each component of an element, for example if they represent different features of a training example. In addition to tuples, you can use `collections.namedtuple` or a dictionary mapping strings to tensors to represent a single element of a `Dataset`.

```
dataset = tf.contrib.data.Dataset.from_tensor_slices(
   {"a": tf.random_uniform([4]),
    "b": tf.random_uniform([4, 100], maxval=100, dtype=tf.int32)})
print(dataset.output_types)  # ==> "{'a': tf.float32, 'b': tf.int32}"
print(dataset.output_shapes)  # ==> "{'a': (), 'b': (100,)}"
```

The `Dataset` transformations support datasets of any structure. When using the `Dataset.map()`, `Dataset.flat_map()`, and `Dataset.filter()` transformations, which apply a function to each element, the element structure determines the arguments of the function:

```
dataset1 = dataset1.map(lambda x: ...)

dataset2 = dataset2.flat_map(lambda x, y: ...)

# Note: Argument destructuring is not available in Python 3.
dataset3 = dataset3.filter(lambda x, (y, z): ...)
```

Creating an iterator

One you have built a `Dataset` to represent your input data, the next step is to create an `Iterator` to access elements from that dataset. The `Dataset` API currently supports three kinds of iterator, in increasing level of sophistication:

- **one-shot**,
- **initializable**,
- **reinitializable**, and
- **feedable**.

A **one-shot** iterator is the simplest form of iterator, which only supports iterating once through a dataset, with no need for explicit initialization. One-shot iterators handle almost all of the cases that the existing queue-based input pipelines support, but they do not support parameterization. Using the example of `Dataset.range()`:

```
dataset = tf.contrib.data.Dataset.range(100)
iterator = dataset.make_one_shot_iterator()
next_element = iterator.get_next()

for i in range(100):
  value = sess.run(next_element)
  assert i == value
```

An **initializable** iterator requires you to run an explicit `iterator.initializer` operation before using it. In exchange for this inconvenience, it enables you to *parameterize* the definition of the dataset, using one or more `tf.placeholder()` tensors that can be fed when you initialize the iterator. Continuing the `Dataset.range()` example:

```
max_value = tf.placeholder(tf.int64, shape=[])
dataset = tf.contrib.data.Dataset.range(max_value)
iterator = dataset.make_initializable_iterator()
next_element = iterator.get_next()

# Initialize an iterator over a dataset with 10 elements.
sess.run(iterator.initializer, feed_dict={max_value: 10})
for i in range(10):
  value = sess.run(next_element)
  assert i == value

# Initialize the same iterator over a dataset with 100 elements.
sess.run(iterator.initializer, feed_dict={max_value: 100})
for i in range(100):
  value = sess.run(next_element)
  assert i == value
```

A **reinitializable** iterator can be initialized from multiple different `Dataset` objects. For example, you might have a training input pipeline that uses random perturbations to the input images to improve generalization, and a validation input pipeline that evaluates predictions on unmodified data. These pipelines will typically use different `Dataset` objects that have the same structure (i.e. the same types and compatible shapes for each component).

```
# Define training and validation datasets with the same structure.
training_dataset = tf.contrib.data.Dataset.range(100).map(
    lambda x: x + tf.random_uniform([], -10, 10, tf.int64))
validation_dataset = tf.contrib.data.Dataset.range(50)

# A reinitializable iterator is defined by its structure. We could use the
# `output_types` and `output_shapes` properties of either `training_dataset`
# or `validation_dataset` here, because they are compatible.
```

```
iterator = Iterator.from_structure(training_dataset.output_types,
                                   training_dataset.output_shapes)
next_element = iterator.get_next()

training_init_op = iterator.make_initializer(training_dataset)
validation_init_op = iterator.make_initializer(validation_dataset)

# Run 20 epochs in which the training dataset is traversed, followed by the
# validation dataset.
for _ in range(20):
  # Initialize an iterator over the training dataset.
  sess.run(training_init_op)
  for _ in range(100):
    sess.run(next_element)

  # Initialize an iterator over the validation dataset.
  sess.run(validation_init_op)
  for _ in range(50):
    sess.run(next_element)
```

A **feedable** iterator can be used together with `tf.placeholder` (https://www.tensorflow.org/api_docs/python/tf/placeholder) to select what `Iterator` to use in each call to `tf.Session.run` (https://www.tensorflow.org/api_docs/python/tf/Session#run), via the familiar `feed_dict` mechanism. It offers the same functionality as a reinitializable iterator, but it does not require you to initialize the iterator from the start of a dataset when you switch between iterators. For example, using the same training and validation example from above, you can use `tf.contrib.data.Iterator.from_string_handle` (https://www.tensorflow.org/api_docs/python/tf/contrib/data/Iterator#from_string_handle) to define a feedable iterator that allows you to switch between the two datasets:

```
# Define training and validation datasets with the same structure.
training_dataset = tf.contrib.data.Dataset.range(100).map(
    lambda x: x + tf.random_uniform([], -10, 10, tf.int64)).repeat()
validation_dataset = tf.contrib.data.Dataset.range(50)

# A feedable iterator is defined by a handle placeholder and its structure. We
# could use the `output_types` and `output_shapes` properties of either
# `training_dataset` or `validation_dataset` here, because they have
# identical structure.
handle = tf.placeholder(tf.string, shape=[])
iterator = tf.contrib.data.Iterator.from_string_handle(
    handle, training_dataset.output_types, training_dataset.output_shapes)
next_element = iterator.get_next()

# You can use feedable iterators with a variety of different kinds of iterato
# (such as one-shot and initializable iterators).
```

```
training_iterator = training_dataset.make_one_shot_iterator()
validation_iterator = validation_dataset.make_initializable_iterator()

# The `Iterator.string_handle()` method returns a tensor that can be evaluate
# and used to feed the `handle` placeholder.
training_handle = sess.run(training_iterator.string_handle())
validation_handle = sess.run(validation_iterator.string_handle())

# Loop forever, alternating between training and validation.
while True:
  # Run 200 steps using the training dataset. Note that the training dataset 
  # infinite, and we resume from where we left off in the previous `while` lo
  # iteration.
  for _ in range(200):
    sess.run(next_element, feed_dict={handle: training_handle})

  # Run one pass over the validation dataset.
  sess.run(validation_iterator.initializer)
  for _ in range(50):
    sess.run(next_element, feed_dict={handle: validation_handle})
```

Consuming values from an iterator

The `Iterator.get_next()` method returns one or more `tf.Tensor` objects that correspond to the symbolic next element of an iterator. Each time these tensors are evaluated, they take the value of the next element in the underlying dataset. (Note that, like other stateful objects in TensorFlow, calling `Iterator.get_next()` does not immediately advance the iterator. Instead you must use the returned `tf.Tensor` objects in a TensorFlow expression, and pass the result of that expression to `tf.Session.run()` to get the next elements and advance the iterator.)

If the iterator reaches the end of the dataset, executing the `Iterator.get_next()` operation will raise a `tf.errors.OutOfRangeError`. After this point the iterator will be in an unusable state, and you must initialize it again if you want to use it further.

```
dataset = tf.contrib.data.Dataset.range(5)
iterator = dataset.make_initializable_iterator()
next_element = iterator.get_next()

# Typically `result` will be the output of a model, or an optimizer's
# training operation.
result = tf.add(next_element, next_element)

sess.run(iterator.initializer)
print(sess.run(result))  # ==> "0"
```

```
print(sess.run(result))  # ==> "2"
print(sess.run(result))  # ==> "4"
print(sess.run(result))  # ==> "6"
print(sess.run(result))  # ==> "8"
try:
  sess.run(result)
except tf.errors.OutOfRangeError:
  print("End of dataset")  # ==> "End of dataset"
```

A common pattern is to wrap the "training loop" in a `try`-`except` block:

```
sess.run(iterator.initializer)
while True:
  try:
    sess.run(result)
  except tf.errors.OutOfRangeError:
    break
```

If each element of the dataset has a nested structure, the return value of `Iterator.get_next()` will be one or more `tf.Tensor` objects in the same nested structure:

```
dataset1 = tf.contrib.data.Dataset.from_tensor_slices(tf.random_uniform([4, 1
dataset2 = tf.contrib.data.Dataset.from_tensor_slices((tf.random_uniform([4])
dataset3 = tf.contrib.data.Dataset.zip((dataset1, dataset2))

iterator = dataset3.make_initializable_iterator()

sess.run(iterator.initializer)
next1, (next2, next3) = iterator.get_next()
```

Note that evaluating *any* of `next1`, `next2`, or `next3` will advance the iterator for all components. A typical consumer of an iterator will include all components in a single expression.

Reading input data

Consuming NumPy arrays

If all of your input data fit in memory, the simplest way to create a `Dataset` from them is to convert them to `tf.Tensor` objects and use `Dataset.from_tensor_slices()`.

```
# Load the training data into two NumPy arrays, for example using `np.load()`.
with np.load("/var/data/training_data.npy") as data:
  features = data["features"]
  labels = data["labels"]

# Assume that each row of `features` corresponds to the same row as `labels`.
assert features.shape[0] == labels.shape[0]

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

Note that the above code snippet will embed the `features` and `labels` arrays in your TensorFlow graph as `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.

As an alternative, you can define the `Dataset` in terms of `tf.placeholder()` tensors, and *feed* the NumPy arrays when you initialize an `Iterator` over the dataset.

```
# Load the training data into two NumPy arrays, for example using `np.load()`.
with np.load("/var/data/training_data.npy") as data:
  features = data["features"]
  labels = data["labels"]

# Assume that each row of `features` corresponds to the same row as `labels`.
assert features.shape[0] == labels.shape[0]

features_placeholder = tf.placeholder(features.dtype, features.shape)
labels_placeholder = tf.placeholder(labels.dtype, labels.shape)

dataset = tf.contrib.data.Dataset.from_tensor_slices((features_placeholder, la
# [Other transformations on `dataset`...]
dataset = ...
iterator = dataset.make_initializable_iterator()

sess.run(iterator.initializer, feed_dict={features_placeholder: features,
                                          labels_placeholder: labels})
```

Consuming TFRecord data

The `Dataset` 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.contrib.data.TFRecordDataset` class enables you to stream over the contents of one or more TFRecord files as part of an input pipeline.

```
# Creates a dataset that reads all of the examples from two files.
filenames = ["/var/data/file1.tfrecord", "/var/data/file2.tfrecord"]
dataset = tf.contrib.data.TFRecordDataset(filenames)
```

The `filenames` argument to the `TFRecordDataset` initializer can either be a string, a list of strings, or a `tf.Tensor` of strings. Therefore if you have two sets of files for training and validation purposes, you can use a `tf.placeholder(tf.string)` to represent the filenames, and initialize an iterator from the appropriate filenames:

```
filenames = tf.placeholder(tf.string, shape=[None])
dataset = tf.contrib.data.TFRecordDataset(filenames)
dataset = dataset.map(...)  # Parse the record into tensors.
dataset = dataset.repeat()  # Repeat the input indefinitely.
dataset = dataset.batch(32)
iterator = dataset.make_initializable_iterator()

# You can feed the initializer with the appropriate filenames for the current
# phase of execution, e.g. training vs. validation.

# Initialize `iterator` with training data.
training_filenames = ["/var/data/file1.tfrecord", "/var/data/file2.tfrecord"]
sess.run(iterator.initializer, feed_dict={filenames: training_filenames})

# Initialize `iterator` with validation data.
validation_filenames = ["/var/data/validation1.tfrecord", ...]
sess.run(iterator.initializer, feed_dict={filenames: validation_filenames})
```

Consuming text data

Many datasets are distributed as one or more text files. The `tf.contrib.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. Like a `TFRecordDataset`, `TextLineDataset` accepts `filenames` as a `tf.Tensor`, so you can parameterize it by passing a `tf.placeholder(tf.string)`.

```
filenames = ["/var/data/file1.txt", "/var/data/file2.txt"]
dataset = tf.contrib.data.TextLineDataset(filenames)
```

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()` and `Dataset.filter()` transformations. To apply these transformations to each file separately, we use `Dataset.flat_map()` to create a nested `Dataset` for each file.

```
filenames = ["/var/data/file1.txt", "/var/data/file2.txt"]

dataset = tf.contrib.data.Dataset.from_tensor_slices(filenames)

# Use `Dataset.flat_map()` to transform each file as a separate nested datase
# and then concatenate their contents sequentially into a single "flat" datas
# * Skip the first line (header row).
# * Filter out lines beginning with "#" (comments).
dataset = dataset.flat_map(
    lambda filename: (
        tf.contrib.data.TextLineDataset(filename)
        .skip(1)
        .filter(lambda line: tf.not_equal(tf.substr(line, 0, 1), "#"))))
```

Preprocessing data with `Dataset.map()`

The `Dataset.map(f)` transformation produces a new dataset by applying a given function `f` to each element of the input dataset. It is based on the map() function (https://en.wikipedia.org/wiki/Map_(higher-order_function)) that is commonly applied to lists (and other structures) in functional programming languages. The function `f` takes the `tf.Tensor` objects that represent a single element in the input, and returns the `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()`.

Parsing `tf.Example` protocol buffer messages

Many input pipelines extract `tf.train.Example` protocol buffer messages from a TFRecord-format file (written, for example, using `tf.python_io.TFRecordWriter`). Each `tf.train.Example` record contains one or more "features", and the input pipeline typically converts these features into tensors.

```
# Transforms a scalar string `example_proto` into a pair of a scalar string a
# a scalar integer, representing an image and its label, respectively.
```

```
def _parse_function(example_proto):
  features = {"image": tf.FixedLenFeature((), tf.string, default_value=""),
              "label": tf.FixedLenFeature((), tf.int32, default_value=0)}
  parsed_features = tf.parse_single_example(example_proto, features)
  return parsed_features["image"], parsed_features["label"]

# Creates a dataset that reads all of the examples from two files, and extrac
# the image and label features.
filenames = ["/var/data/file1.tfrecord", "/var/data/file2.tfrecord"]
dataset = tf.contrib.data.TFRecordDataset(filenames)
dataset = dataset.map(_parse_function)
```

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.

```
# Reads an image from a file, decodes it into a dense tensor, and resizes it
# to a fixed shape.
def _parse_function(filename, label):
  image_string = tf.read_file(filename)
  image_decoded = tf.image.decode_image(image_string)
  image_resized = tf.image.resize_images(image_decoded, [28, 28])
  return image_resized, label

# A vector of filenames.
filenames = tf.constant(["/var/data/image1.jpg", "/var/data/image2.jpg", ...]

# `labels[i]` is the label for the image in `filenames[i].
labels = tf.constant([0, 37, ...])

dataset = tf.contrib.data.Dataset.from_tensor_slices((filenames, labels))
dataset = dataset.map(_parse_function)
```

Applying arbitrary Python logic with `tf.py_func()`

For performance reasons, we encourage you to use TensorFlow operations for preprocessing your data whenever possible. However, it is sometimes useful to call upon external Python libraries when parsing your input data. To do so, invoke, the `tf.py_func()` operation in a `Dataset.map()` transformation.

```
import cv2
```

```
# Use a custom OpenCV function to read the image, instead of the standard
# TensorFlow `tf.read_file()` operation.
def _read_py_function(filename, label):
  image_decoded = cv2.imread(image_string, cv2.IMREAD_GRAYSCALE)
  return image_decoded, label

# Use standard TensorFlow operations to resize the image to a fixed shape.
def _resize_function(image_decoded, label):
  image_decoded.set_shape([None, None, None])
  image_resized = tf.image.resize_images(image_decoded, [28, 28])
  return image_resized, label

filenames = ["/var/data/image1.jpg", "/var/data/image2.jpg", ...]
labels = [0, 37, 29, 1, ...]

dataset = tf.contrib.data.Dataset.from_tensor_slices((filenames, labels))
dataset = dataset.map(
    lambda filename, label: tf.py_func(
        _read_py_function, [filename, label], [tf.uint8, label.dtype]))
dataset = dataset.map(_resize_function)
```

Batching dataset elements

Simple batching

The simplest form of batching stacks `n` consecutive elements of a dataset into a single element. The `Dataset.batch()` transformation does exactly this, with the same constraints as the `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.contrib.data.Dataset.range(100)
dec_dataset = tf.contrib.data.Dataset.range(0, -100, -1)
dataset = tf.contrib.data.Dataset.zip((inc_dataset, dec_dataset))
batched_dataset = dataset.batch(4)

iterator = batched_dataset.make_one_shot_iterator()
next_element = iterator.get_next()

print(sess.run(next_element))  # ==> ([0, 1, 2,   3],   [ 0, -1,  -2,  -3])
print(sess.run(next_element))  # ==> ([4, 5, 6,   7],   [-4, -5,  -6,  -7])
print(sess.run(next_element))  # ==> ([8, 9, 10, 11],   [-8, -9, -10, -11])
```

Batching tensors with padding

The above recipe works for tensors that all have the same size. However, many models (e.g. sequence models) work with input data that can have varying size (e.g. sequences of different lengths). To handle this case, the `Dataset.padded_batch()` transformation enables you to batch tensors of different shape by specifying one or more dimensions in which they may be padded.

```
dataset = tf.contrib.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])

iterator = dataset.make_one_shot_iterator()
next_element = iterator.get_next()

print(sess.run(next_element))  # ==> [[0, 0, 0], [1, 0, 0], [2, 2, 0], [3, 3,
print(sess.run(next_element))  # ==> [[4, 4, 4, 4, 0, 0, 0],
                               #      [5, 5, 5, 5, 5, 0, 0],
                               #      [6, 6, 6, 6, 6, 6, 0],
                               #      [7, 7, 7, 7, 7, 7, 7]]
```

The `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 `Dataset` 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()` transformation. For example, to create a dataset that repeats its input for 10 epochs:

```
filenames = ["/var/data/file1.tfrecord", "/var/data/file2.tfrecord"]
dataset = tf.contrib.data.TFRecordDataset(filenames)
dataset = dataset.map(...)
dataset = dataset.repeat(10)
dataset = dataset.batch(32)
```

Applying the `Dataset.repeat()` transformation with no arguments will repeat the input indefinitely. The `Dataset.repeat()` transformation concatenates its arguments without signaling the end of one epoch and the beginning of the next epoch.

If you want to receive a signal at the end of each epoch, you can write a training loop that catches the `tf.errors.OutOfRangeError` at the end of a dataset. At that point you might collect some statistics (e.g. the validation error) for the epoch.

```
filenames = ["/var/data/file1.tfrecord", "/var/data/file2.tfrecord"]
dataset = tf.contrib.data.TFRecordDataset(filenames)
dataset = dataset.map(...)
dataset = dataset.batch(32)
iterator = dataset.make_initializable_iterator()
next_element = iterator.get_next()

# Compute for 100 epochs.
for _ in range(100):
  sess.run(iterator.initializer)
  while True:
    try:
      sess.run(next_element)
    except tf.errors.OutOfRangeError:
      break

  # [Perform end-of-epoch calculations here.]
```

Randomly shuffling input data

The `Dataset.shuffle()` transformation randomly shuffles the input dataset using a similar algorithm to `tf.RandomShuffleQueue`: it maintains a fixed-size buffer and chooses the next element uniformly at random from that buffer.

```
filenames = ["/var/data/file1.tfrecord", "/var/data/file2.tfrecord"]
dataset = tf.contrib.data.TFRecordDataset(filenames)
dataset = dataset.map(...)
dataset = dataset.shuffle(buffer_size=10000)
dataset = dataset.batch(32)
dataset = dataset.repeat()
```

Using high-level APIs

The `tf.train.MonitoredTrainingSession` (https://www.tensorflow.org/api_docs/python/tf/train/MonitoredTrainingSession) API simplifies many aspects of running TensorFlow in a distributed setting. `MonitoredTrainingSession` uses the `tf.errors.OutOfRangeError` (https://www.tensorflow.org/api_docs/python/tf/errors/OutOfRangeError) to signal that training has completed, so to use it with the `Dataset` API, we recommend using `Dataset.make_one_shot_iterator()`. For example:

```
filenames = ["/var/data/file1.tfrecord", "/var/data/file2.tfrecord"]
dataset = tf.contrib.data.TFRecordDataset(filenames)
dataset = dataset.map(...)
dataset = dataset.shuffle(buffer_size=10000)
dataset = dataset.batch(32)
dataset = dataset.repeat(num_epochs)
iterator = dataset.make_one_shot_iterator()

next_example, next_label = iterator.get_next()
loss = model_function(next_example, next_label)

training_op = tf.train.AdagradOptimizer(...).minimize(loss)

with tf.train.MonitoredTrainingSession(...) as sess:
  while not sess.should_stop():
    sess.run(training_op)
```

To use a `Dataset` in the `input_fn` of a [`tf.estimator.Estimator`](https://www.tensorflow.org/api_docs/python/tf/estimator/Estimator) (https://www.tensorflow.org/api_docs/python/tf/estimator/Estimator), we also recommend using `Dataset.make_one_shot_iterator()`. For example:

```
def dataset_input_fn():
  filenames = ["/var/data/file1.tfrecord", "/var/data/file2.tfrecord"]
  dataset = tf.contrib.data.TFRecordDataset(filenames)

  # Use `tf.parse_single_example()` to extract data from a `tf.Example`
  # protocol buffer, and perform any additional per-record preprocessing.
  def parser(record):
    keys_to_features = {
        "image_data": tf.FixedLenFeature((), tf.string, default_value=""),
        "date_time": tf.FixedLenFeature((), tf.int64, default_value=""),
        "label": tf.FixedLenFeature((), tf.int64,
                                    default_value=tf.zeros([], dtype=tf.int64
    }
    parsed = tf.parse_single_example(record, keys_to_features)

    # Perform additional preprocessing on the parsed data.
    image = tf.decode_jpeg(parsed["image_data"])
    image = tf.reshape(image, [299, 299, 1])
    label = tf.cast(parsed["label"], tf.int32)

    return {"image_data": image, "date_time": parsed["date_time"]}, label
```

```
# Use `Dataset.map()` to build a pair of a feature dictionary and a label
# tensor for each example.
dataset = dataset.map(parser)
dataset = dataset.shuffle(buffer_size=10000)
dataset = dataset.batch(32)
dataset = dataset.repeat(num_epochs)
iterator = dataset.make_one_shot_iterator()

# `features` is a dictionary in which each value is a batch of values for
# that feature; `labels` is a batch of labels.
features, labels = iterator.get_next()
return features, labels
```

Last updated 八月 17, 2017.