tensorflow中的dataset

Reading time ~8 minutes

1.数据导入

tf.data API可以让你以简单可复用的方式构建复杂的Input Pipeline。例如：一个图片模型的Pipeline可能会聚合在一个分布式文件系统中的多个文件，对每个图片进行随机扰动（random perturbations），接着将随机选中的图片合并到一个training batch中。一个文本模型的Pipeline可能涉及到：从原始文本数据中抽取特征，将它们通过一个lookup table转换成embedding identifiers，然后将不同的长度序列batch在一起。tf.data API可以很方便地以不同的数据格式处理大量的数据，以及处理复杂的转换。

Dataset API引入了两个新的抽象类到Tensorflow中：

- **tf.data.Dataset**：表示一串元素（elements），其中每个元素包含了一或多个Tensor对象。例如：在一个图片pipeline中，一个元素可以是单个训练样本，它们带有一个表示图片数据的tensors和一个label组成的pair。**有两种不同的方式创建一个dataset**：
  - 创建一个**source** (例如：Dataset.from_tensor_slices())，从一或多个tf.Tensor对象中构建一个dataset
  - 应用一个**transformation**（例如：Dataset.batch()），从一或多个tf.data.Dataset对象上构建一个dataset
- **tf.data.Iterator**：它提供了主要的方式来从一个dataset中抽取元素。通过Iterator.get_next()返回的该操作会yields出Datasets中的下一个元素，作为输入pipeline和模型间的接口使用。最简单的iterator是一个“one-shot iterator”，它与一个指定的Dataset相关联，通过它来进行迭代。对于更复杂的使用，Iterator.initializer操作可以使用不同的datasets重新初始化（reinitialize）和参数化（parameterize）一个iterator，例如，在同一个程序中通过training data和validation data迭代多次。

2.基本机制

这部分描述了创建不同Dataset和Iterator对象的机制，以及如何使用它们来抽取数据。

要想启动一个input pipeline，你必须定义一个source。例如，为了从内存中的一些tensors构建一个Dataset，你可以使用tf.data.Dataset.from_tensors() 以及tf.data.Dataset.from_tensor_slices()。另一种方法，如果你的输入数据在磁盘上以推荐的TFRecord格式存储，你可以构建一个

tf.data.TFRecordDataset。一旦你有一个Dataset对象，通过在tf.data.Dataset对象上链式方法调用，你可以将它转化成一个新的Dataset。例如，你可以使用per-element transformations，比如：Dataset.map()，（它会在每个元素上应用一个function），以及multi-element transformations，比如：Dataset.batch()。更多详见api

从一个Dataset上消费values的最常用方法，是生成一个iterator对象，它提供了一次可以访问dataset中的一个元素（例如：通过调用Dataset.make_one_shot_iterator()）。**tf.data.Iterator提供了两个操作**：

- Iterator.initializer：它允许你(re)initialize iterator的状态
- Iterator.get_next()：它返回tf.Tensor对象，对应于指定的下一个元素。

2.1 Dataset结构

一个dataset由element组成，它们每个都具有相同的结构。一个元素包含了一或多个tf.Tensor对象，称为“components“。每个component都具有一个**tf.DType**：它表示在tensor中的元素的类型；以及一个**tf.TensorShape**：它表示每个元素的静态shape。Dataset.output_types 和 Dataset.output_shapes 属性允许你观察到一个dataset元素的每个component内省的types和shapes。这些属性的这种嵌套式结构（nested structure），映射到一个元素（它可以是单个tensor、一个tensors的tuple、一个tensors的嵌套式tuple）的结构上。例如：

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

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

为一个元素(element)的每个component给定names很方便，例如，如果它们表示一个训练样本的不同features。除了tuples，你可以使用collections.namedtuple，或者一个将strings映射为关于tensors的字典来表示一个Dataset的单个元素。

```
dataset = tf.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,)}"
```

Dataset的转换（transformations）支持任何结构的datasets。当使用Dataset.map()，Dataset.flat_map()，以及Dataset.filter()转换时，它们会对每个element应用一个function，元素结构决定了函数的参数：

```
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): ...)
```

2.2 创建一个iterator

一旦你已经构建了一个Dataset来表示你的输入数据，下一步是创建一个Iterator来访问dataset的elements。Dataset API当前支持四种iterator，复杂度依次递增：

- **one-shot**
- **initializable**
- **reinitializable**
- **feedable**

one-shot iterator是最简单的iterator，它只支持在一个dataset上迭代一次的操作，不需要显式初始化。One-shot iterators可以处理几乎所有的已存在的基于队列的input pipeline支持的情况，但它们不支持参数化（parameterization）。使用Dataset.range()示例如下：

```
dataset = tf.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
```

initializable iterator在使用它之前需要你返回一个显式的iterator.initializer操作。虽然有些不便，但它允许你可以对dataset的定义进行参数化（parameterize），使用一或多个tf.placeholder() tensors：它们可以当你初始化iterator时被feed进去。继续Dataset.range() 的示例：

```
max_value = tf.placeholder(tf.int64, shape=[])
dataset = tf.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
```

reinitializable iterator可以从多个不同的Dataset对象处初始化。例如，你可能有一个training input pipeline（它对输入图片做随机扰动来提高泛化能力）；以及一个validation input pipeline（它会在未修改过的数据上进行预测的评估）。这些pipeline通常使用不同的Dataset对象，但它们具有相同的结构（例如：对每个component相同的types和shapes）

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

feedable iterator可以与tf.placeholder一起使用，通过熟悉的feed_dict机制，来选择在每次调用tf.Session.run所使用的Iterator，。它提供了与reinitializable iterator相同的功能，但当你在iterators间相互切换时，它不需要你去初始化iterator。例如：使用上述相同的training和

validation样本，你可以使用tf.data.Iterator.from_string_handle来定义一个feedable iterator，并允许你在两个datasets间切换：

```
# Define training and validation datasets with the same structure.
training_dataset = tf.data.Dataset.range(100).map(
    lambda x: x + tf.random_uniform([], -10, 10, tf.int64)).repeat()
validation_dataset = tf.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.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 iterator
# (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 evaluated
# 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 is
  # infinite, and we resume from where we left off in the previous `while` loop
  # 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})
```

2.3 从一个iterator上消费values

Iterator.get_next()方法会返回一或多个tf.Tensor对象，对应于一个iterator的下一个element。每次这些tensors被评测时，它们会在底层的dataset中获得下一个element的value。（注意：类似于Tensorflow中其它的有状态对象，调用Iterator.get_next()不会立即让iterator前移。相反的，你必须使用Tensorflow表达式所返回的tf.Tensor对象，传递该表达式的结果给tf.Session.run()，来获取下一个elements，并让iterator前移）

如果iterator达到了dataset的结尾，执行Iterator.get_next()操作会抛出一个tf.errors.OutOfRangeError。在这之后，iterator会以一个不可用的状态存在，如果你想进一步使

用必须重新初始化它。

```
dataset = tf.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"
```

一种常用的模式是，将"training loop"封装到一个try-except块中：

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

如果dataset的每个元素都具有一个嵌套的结构，Iterator.get_next()的返回值将会是以相同嵌套结构存在的一或多个tf.Tensor对象：

```
dataset1 = tf.data.Dataset.from_tensor_slices(tf.random_uniform([4, 10]))
dataset2 = tf.data.Dataset.from_tensor_slices((tf.random_uniform([4]), tf.random_uniform([4, 100])))
dataset3 = tf.data.Dataset.zip((dataset1, dataset2))

iterator = dataset3.make_initializable_iterator()

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

注意，对next1, next2, or next3的任意一个进行评估都会为所有components进行iterator。一个iterator的一种常见consumer将包含在单个表达式中的所有components。

3.读取输入数据

3.1 消费Numpy arrays

如果所有的输入数据都加载进内存，最简单的方式是，从输入数据中创建一个Dataset，并将它们转换成tf.Tensor对象，并使用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.data.Dataset.from_tensor_slices((features, labels))
```

注意，上述的代码段会将features arrays和labels arrays作为tf.constant() 操作嵌套进你的TensorFlow graph中。这在小数据集上能良好运行，但会浪费内存——因为array的内存会被拷贝多次————对于tf.GraphDef的protocol buffer，只可以运行2GB的内存限制。

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

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

3.2 消费TFRecord数据

Dataset API支持多种文件格式，因此你可以处理超过内存大小的大数据集。例如，TFRecord文件格式是一种简单的面向记录的二进制格式，许多TensorFlow应用都用它来做训练数据。tf.data.TFRecordDataset类允许你在一或多个TFRecord文件的内容上进行流化，将它们作为input pipeline的一部分：

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

TFRecordDataset initializer的**filenames**参数，可以是一个string，也可以是一列string，或者关于strings的一个tf.Tensor。因此，如果你具有两个文件集合，分别对应训练数据和验证数据，你可以使用一个tf.placeholder(tf.string)来表示filenames，并从合适的filenames上初始化一个iterator：

```
filenames = tf.placeholder(tf.string, shape=[None])
dataset = tf.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})
```

3.3 消费文本数据

许多datasets以一或多个文本文件分布。tf.data.TextLineDataset提供了一种简单的方式来从文本文件中抽取行(lines)。给定一或多个filenames，一个TextLineDataset将为这些文件的每行生成一个string型的element。与TFRecordDataset类似，TextLineDataset会接受filenames参数作为一个tf.Tensor，因此你可以通过传递一个tf.placeholder(tf.string)对它参数化。

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

缺省的，一个TextLineDataset会yields每个文件的所有行，这不是我们所希望的，例如，如果该文件使用一个header line开始，或包含注释。这些行通过Dataset.skip() 和 Dataset.filter() 转换被移去。为了将这些转换独立地应用每个文件上，我们使用Dataset.flat_map() 来为每个文件创建一个嵌套的Dataset。

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

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

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

4.使用Dataset.map()预处理数据

通过在输入数据集的每个element上应用一个给定的函数f，Dataset.map(f)变换会产生一个新的dataset。该函数f会接受tf.Tensor对象（它表示input中的单个element）作为参数，并返回tf.Tensor对象（它表示在new dataset中的单个element）。它的实现使用了标准的TensorFlow操作来将一个element转换成另一个。

本节包含了如何使用Dataset.map()的示例。

4.1 解析tf.Example protocol buffer messages

许多input pipelines会从一个TFRecord格式的文件中抽取tf.train.Example protocol buffer messages（例如：使用tf.python_io.TFRecordWriter）。每个tf.train.Example record包含一或多个"features"，input pipeline通常会将这些features转换成tensors。

```
# Transforms a scalar string `example_proto` into a pair of a scalar string and
# 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 extracts
# the image and label features.
filenames = ["/var/data/file1.tfrecord", "/var/data/file2.tfrecord"]
dataset = tf.data.TFRecordDataset(filenames)
dataset = dataset.map(_parse_function)
```

4.2 将图片数据进行decoding，并resizing

当对真实世界的图片数据训练一个神经网络时，经常需要将不同size的图片转换成同一size，因此，必须批量转换成一个固定的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.data.Dataset.from_tensor_slices((filenames, labels))
dataset = dataset.map(_parse_function)
```

4.3 使用tf.py_func()

出于性能的原因，我们鼓励你去使用TensorFlow operations来预处理数据。然而，有时，当解析你的输入数据时调用额外的python库会很有用。可以通过在一个Dataset.map() 转换上调用 tf.py_func() operation来达到这一点。

```
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.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)
```

5.打包元素（Batching dataset elements）

5.1 简单的batching

batching的最简单方式是，将数据集上n个连续的elements进行stack成单个elements。Dataset.batch() 转换可以精准地做到这一点，它使用与tf.stack() 操作相同的constraints，应用在元素的每个component上：例如，对于每个元素i，所有元素必须具有一个相同shape的tensor：

```
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)

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])
```

5.2 使用padding打包tensors

上面的方法需要相同的size。然而，许多模型（比如：序列模型）的输入数据的size多种多样（例如：序列具有不同的长度）为了处理这种情况，Dataset.padded_batch() 转换允许你将不同shape的tensors进行batch，通过指定一或多个dimensions，在其上进行pad。

```
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])

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, 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]]
```

Dataset.padded_batch() 转换允许你为每个component的每个dimension设置不同的padding，它可以是可变的长度（在样本上指定None即可）或恒定长度。你可以对padding值（缺省为0.0）进行override。

6.训练工作流（Training workflows）

6.1 处理多个epochs

Dataset API提供了两种主要方式来处理相同数据的多个epochs。

最简单的方式是，在一个dataset上使用Dataset.repeat()转换进行多轮迭代。例如：创建一个dataset，并repeat它的输入10个epochs。

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

使用无参数的Dataset.repeat() 会不断重复input 。Dataset.repeat() 转换将它的参数进行连接，无需一轮的结束处以及下一轮的开始处发出信号。

如果你想在每一轮的结尾接收到一个信号，你可以编写一个training loop，在dataset的结尾处捕获tf.errors.OutOfRangeError。在那时刻，你可以收集到该轮的一些统计信息（例如：validation error）

```
filenames = ["/var/data/file1.tfrecord", "/var/data/file2.tfrecord"]
dataset = tf.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.]
```

6.2 对输入数据进行random shuffling

Dataset.shuffle() 转换会与tf.RandomShuffleQueue使用相同的算法对输入数据集进行随机shuffle：它会维持一个固定大小的buffer，并从该buffer中随机均匀地选择下一个元素：

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

6.3 使用高级API

tf.train.MonitoredTrainingSession API可以简化分布式设置下运行的Tensorflow的许多方面。当训练完成时，MonitoredTrainingSession使用 tf.errors.OutOfRangeError来发射信号，因此为了配合Dataset API使用它，我们推荐使用Dataset.make_one_shot_iterator()。例如：

```
filenames = ["/var/data/file1.tfrecord", "/var/data/file2.tfrecord"]
dataset = tf.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)
```

为了在tf.estimator.Estimator的input_fn使用一个Dataset，我们推荐使用Dataset.make_one_shot_iterator()。例如：

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

参考

官方tensorflow datasets