tf.data.TextLineDataset

Contents

Class **TextLineDataset**

Inherits From: `Dataset`

Defined in `tensorflow/python/data/ops/readers.py`.

A `Dataset` comprising lines from one or more text files.

Properties

`output_shapes`

`output_types`

Methods

`__init__`

```
__init__(
    filenames,
    compression_type=None,
    buffer_size=None
)
```

Creates a `TextLineDataset`.

Args:

- `filenames`: A `tf.string` tensor containing one or more filenames.
- `compression_type`: (Optional.) A `tf.string` scalar evaluating to one of `""` (no compression), `"ZLIB"`, or `"GZIP"`.
- `buffer_size`: (Optional.) A `tf.int64` scalar denoting the number of bytes to buffer. A value of 0 results in the default buffering values chosen based on the compression type.

`apply`

```
apply(transformation_func)
```

Apply a transformation function to this dataset.

`apply` enables chaining of custom `Dataset` transformations, which are represented as functions that take one `Dataset` argument and return a transformed `Dataset`.

For example:

```
dataset = (dataset.map(lambda x: x ** 2)
           .apply(group_by_window(key_func, reduce_func, window_size))
           .map(lambda x: x ** 3))
```

Args:

- `transformation_func`: A function that takes one `Dataset` argument and returns a `Dataset`.

Returns:

The `Dataset` returned by applying `transformation_func` to this dataset.

batch

```
batch(batch_size)
```

Combines consecutive elements of this dataset into batches.

Args:

- `batch_size`: A `tf.int64` scalar `tf.Tensor`, representing the number of consecutive elements of this dataset to combine in a single batch.

Returns:

A `Dataset`.

cache

```
cache(filename='')
```

Caches the elements in this dataset.

Args:

- `filename`: A `tf.string` scalar `tf.Tensor`, representing the name of a directory on the filesystem to use for caching tensors in this Dataset. If a filename is not provided, the dataset will be cached in memory.

Returns:

A `Dataset`.

concatenate

```
concatenate(dataset)
```

Creates a `Dataset` by concatenating given dataset with this dataset.

```
# NOTE: The following examples use `{ ... }` to represent the
# contents of a dataset.
a = { 1, 2, 3 }
b = { 4, 5, 6, 7 }

# Input dataset and dataset to be concatenated should have same
# nested structures and output types.
# c = { (8, 9), (10, 11), (12, 13) }
# d = { 14.0, 15.0, 16.0 }
# a.concatenate(c) and a.concatenate(d) would result in error.

a.concatenate(b) == { 1, 2, 3, 4, 5, 6, 7 }
```

Args:

- `dataset`: `Dataset` to be concatenated.

Returns:

A `Dataset`.

`filter`

```
filter(predicate)
```

Filters this dataset according to `predicate`.

Args:

- `predicate`: A function mapping a nested structure of tensors (having shapes and types defined by `self.output_shapes` and `self.output_types`) to a scalar `tf.bool` tensor.

Returns:

A `Dataset`.

`flat_map`

```
flat_map(map_func)
```

Maps `map_func` across this dataset and flattens the result.

Args:

- `map_func`: A function mapping a nested structure of tensors (having shapes and types defined by `self.output_shapes` and `self.output_types`) to a `Dataset`.

Returns:

A `Dataset`.

`from_generator`

```
from_generator(
    generator,
    output_types,
    output_shapes=None
)
```

Creates a `Dataset` whose elements are generated by `generator`.

The `generator` argument must be a callable object that returns an object that support the `iter()` protocol (e.g. a generator function). The elements generated by `generator` must be compatible with the given `output_types` and (optional) `output_shapes` arguments.

For example:

```
import itertools

def gen():
  for i in itertools.count(1):
    yield (i, [1] * i)

ds = Dataset.from_generator(
    gen, (tf.int64, tf.int64), (tf.TensorShape([]), tf.TensorShape([None])))
value = ds.make_one_shot_iterator().get_next()

sess.run(value)  # (1, array([1]))
sess.run(value)  # (2, array([1, 1]))
```

Args:

- `generator`: A callable object that takes no arguments and returns an object that supports the `iter()` protocol.
- `output_types`: A nested structure of `tf.DType` objects corresponding to each component of an element yielded by `generator`.
- `output_shapes`: (Optional.) A nested structure of `tf.TensorShape` objects corresponding to each component of an element yielded by `generator`.

Returns:

A `Dataset`.

`from_sparse_tensor_slices`

```
from_sparse_tensor_slices(sparse_tensor)
```

Splits each rank-N `tf.SparseTensor` in this dataset row-wise.

Args:

- `sparse_tensor`: A `tf.SparseTensor`.

Returns:

A `Dataset` of rank-(N-1) sparse tensors.

`from_tensor_slices`

```
from_tensor_slices(tensors)
```

Creates a `Dataset` whose elements are slices of the given tensors.

Args:

- `tensors`: A nested structure of tensors, each having the same size in the 0th dimension.

Returns:

A `Dataset`.

`from_tensors`

```
from_tensors(tensors)
```

Creates a `Dataset` with a single element, comprising the given tensors.

Args:

- `tensors`: A nested structure of tensors.

Returns:

A `Dataset`.

`interleave`

```
interleave(
    map_func,
    cycle_length,
    block_length=1
)
```

Maps `map_func` across this dataset, and interleaves the results.

For example, you can use `Dataset.interleave()` to process many input files concurrently:

```
# Preprocess 4 files concurrently, and interleave blocks of 16 records from
# each file.
filenames = ["/var/data/file1.txt", "/var/data/file2.txt", ..."]
dataset = (Dataset.from_tensor_slices(filenames)
           .interleave(lambda x:
               TextLineDataset(x).map(parse_fn, num_parallel_calls=1),
               cycle_length=4, block_length=16))
```

The `cycle_length` and `block_length` arguments control the order in which elements are produced. `cycle_length` controls the number of input elements that are processed concurrently. If you set `cycle_length` to 1, this transformation will handle one input element at a time, and will produce identical results = to `tf.data.Dataset.flat_map`. In general, this transformation will apply `map_func` to `cycle_length` input elements, open iterators on the returned `Dataset` objects, and cycle through them producing `block_length` consecutive elements from each iterator, and consuming the next input element each time it reaches the end of an iterator.

For example:

```
# NOTE: The following examples use `{ ... }` to represent the
# contents of a dataset.
a = { 1, 2, 3, 4, 5 }

# NOTE: New lines indicate "block" boundaries.
a.interleave(lambda x: Dataset.from_tensors(x).repeat(6),
             cycle_length=2, block_length=4) == {
    1, 1, 1, 1,
    2, 2, 2, 2,
    1, 1,
    2, 2,
    3, 3, 3, 3,
    4, 4, 4, 4,
    3, 3,
    4, 4,
    5, 5, 5, 5,
    5, 5,
}
```

NOTE: The order of elements yielded by this transformation is deterministic, as long as `map_func` is a pure function. If `map_func` contains any stateful operations, the order in which that state is accessed is undefined.

Args:

- `map_func`: A function mapping a nested structure of tensors (having shapes and types defined by `self.output_shapes` and `self.output_types`) to a `Dataset`.
- `cycle_length`: The number of elements from this dataset that will be processed concurrently.
- `block_length`: The number of consecutive elements to produce from each input element before cycling to another input element.

Returns:

A `Dataset`.

`list_files`

```
list_files(file_pattern)
```

A dataset of all files matching a pattern.

Example: If we had the following files on our filesystem: - /path/to/dir/a.txt - /path/to/dir/b.py - /path/to/dir/c.py If we pass "/path/to/dir/*.py" as the directory, the dataset would produce: - /path/to/dir/b.py - /path/to/dir/c.py

Args:

- `file_pattern`: A string or scalar string `tf.Tensor`, representing the filename pattern that will be matched.

Returns:

A `Dataset` of strings corresponding to file names.

`make_initializable_iterator`

```
make_initializable_iterator(shared_name=None)
```

Creates an `Iterator` for enumerating the elements of this dataset.

> **Note:** The returned iterator will be in an uninitialized state, and you must run the `iterator.initializer` operation before using it:

```
dataset = ...
iterator = dataset.make_initializable_iterator()
# ...
sess.run(iterator.initializer)
```

Args:

- `shared_name` : (Optional.) If non-empty, the returned iterator will be shared under the given name across multiple sessions that share the same devices (e.g. when using a remote server).

Returns:

An `Iterator` over the elements of this dataset.

`make_one_shot_iterator`

```
make_one_shot_iterator()
```

Creates an `Iterator` for enumerating the elements of this dataset.

N.B. The returned iterator will be initialized automatically. A "one-shot" iterator does not currently support re-initialization.

Returns:

An `Iterator` over the elements of this dataset.

`map`

```
map(
    map_func,
    num_parallel_calls=None
)
```

Maps `map_func` across this datset.

Args:

- `map_func` : A function mapping a nested structure of tensors (having shapes and types defined by `self.output_shapes` and `self.output_types`) to another nested structure of tensors.
- `num_parallel_calls` : (Optional.) A `tf.int32` scalar `tf.Tensor` , representing the number elements to process in

parallel. If not specified, elements will be processed sequentially.

Returns:

A `Dataset`.

`padded_batch`

```
padded_batch(
    batch_size,
    padded_shapes,
    padding_values=None
)
```

Combines consecutive elements of this dataset into padded batches.

Like `Dataset.dense_to_sparse_batch()`, this method combines multiple consecutive elements of this dataset, which might have different shapes, into a single element. The tensors in the resulting element have an additional outer dimension, and are padded to the respective shape in `padded_shapes`.

Args:

- `batch_size`: A `tf.int64` scalar `tf.Tensor`, representing the number of consecutive elements of this dataset to combine in a single batch.
- `padded_shapes`: A nested structure of `tf.TensorShape` or `tf.int64` vector tensor-like objects representing the shape to which the respective component of each input element should be padded prior to batching. Any unknown dimensions (e.g. `tf.Dimension(None)` in a `tf.TensorShape` or `-1` in a tensor-like object) will be padded to the maximum size of that dimension in each batch.
- `padding_values`: (Optional.) A nested structure of scalar-shaped `tf.Tensor`, representing the padding values to use for the respective components. Defaults are `0` for numeric types and the empty string for string types.

Returns:

A `Dataset`.

`prefetch`

```
prefetch(buffer_size)
```

Creates a `Dataset` that prefetches elements from this dataset.

Args:

- `buffer_size`: A `tf.int64` scalar `tf.Tensor`, representing the maximum number elements that will be buffered when prefetching.

Returns:

A `Dataset`.

`range`

```
range(*args)
```

Creates a `Dataset` of a step-separated range of values.

For example:

```
Dataset.range(5) == [0, 1, 2, 3, 4]
Dataset.range(2, 5) == [2, 3, 4]
Dataset.range(1, 5, 2) == [1, 3]
Dataset.range(1, 5, -2) == []
Dataset.range(5, 1) == []
Dataset.range(5, 1, -2) == [5, 3]
```

Args:

- `*args`: follow same semantics as python's xrange. len(args) == 1 -> start = 0, stop = args[0], step = 1 len(args) == 2 -> start = args[0], stop = args[1], step = 1 len(args) == 3 -> start = args[0], stop = args[1, stop = args[2]

Returns:

A `RangeDataset`.

Raises:

- `ValueError`: if len(args) == 0.

`repeat`

```
repeat(count=None)
```

Repeats this dataset `count` times.

Args:

- `count`: (Optional.) A `tf.int64` scalar `tf.Tensor`, representing the number of times the elements of this dataset should be repeated. The default behavior (if `count` is `None` or `-1`) is for the elements to be repeated indefinitely.

Returns:

A `Dataset`.

`shard`

```
shard(
    num_shards,
    index
)
```

Creates a `Dataset` that includes only 1/`num_shards` of this dataset.

This dataset operator is very useful when running distributed training, as it allows each worker to read a unique subset.

When reading a single input file, you can skip elements as follows:

```
d = tf.data.TFRecordDataset(FLAGS.input_file)
d = d.shard(FLAGS.num_workers, FLAGS.worker_index)
d = d.repeat(FLAGS.num_epochs)
d = d.shuffle(FLAGS.shuffle_buffer_size)
d = d.map(parser_fn, num_parallel_calls=FLAGS.num_map_threads)
```

Important caveats:

- Be sure to shard before you use any randomizing operator (such as shuffle).
- Generally it is best if the shard operator is used early in the dataset pipeline. For example, when reading from a set of TFRecord files, shard before converting the dataset to input samples. This avoids reading every file on every worker. The following is an example of an efficient sharding strategy within a complete pipeline:

```
d = Dataset.list_files(FLAGS.pattern)
d = d.shard(FLAGS.num_workers, FLAGS.worker_index)
d = d.repeat(FLAGS.num_epochs)
d = d.shuffle(FLAGS.shuffle_buffer_size)
d = d.repeat()
d = d.interleave(tf.data.TFRecordDataset,
                 cycle_length=FLAGS.num_readers, block_length=1)
d = d.map(parser_fn, num_parallel_calls=FLAGS.num_map_threads)
```

Args:

- `num_shards`: A **`tf.int64`** scalar **`tf.Tensor`**, representing the number of shards operating in parallel.
- `index`: A **`tf.int64`** scalar **`tf.Tensor`**, representing the worker index.

Returns:

A **`Dataset`**.

Raises:

- `ValueError`: if **`num_shards`** or **`index`** are illegal values. Note: error checking is done on a best-effort basis, and aren't guaranteed to be caught upon dataset creation. (e.g. providing in a placeholder tensor bypasses the early checking, and will instead result in an error during a session.run call.)

`shuffle`

```
shuffle(
    buffer_size,
    seed=None,
    reshuffle_each_iteration=None
)
```

Randomly shuffles the elements of this dataset.

Args:

- `buffer_size`: A **`tf.int64`** scalar **`tf.Tensor`**, representing the number of elements from this dataset from which the new dataset will sample.
- `seed`: (Optional.) A **`tf.int64`** scalar **`tf.Tensor`**, representing the random seed that will be used to create the distribution. See **`tf.set_random_seed`** for behavior.

- `reshuffle_each_iteration`: (Optional.) A boolean, which if true indicates that the dataset should be pseudorandomly reshuffled each time it is iterated over. (Defaults to `True`.)

Returns:

A `Dataset`.

`skip`

```
skip(count)
```

Creates a `Dataset` that skips `count` elements from this dataset.

Args:

- `count`: A `tf.int64` scalar `tf.Tensor`, representing the number of elements of this dataset that should be skipped to form the new dataset. If `count` is greater than the size of this dataset, the new dataset will contain no elements. If `count` is -1, skips the entire dataset.

Returns:

A `Dataset`.

`take`

```
take(count)
```

Creates a `Dataset` with at most `count` elements from this dataset.

Args:

- `count`: A `tf.int64` scalar `tf.Tensor`, representing the number of elements of this dataset that should be taken to form the new dataset. If `count` is -1, or if `count` is greater than the size of this dataset, the new dataset will contain all elements of this dataset.

Returns:

A `Dataset`.

`zip`

```
zip(datasets)
```

Creates a `Dataset` by zipping together the given datasets.

This method has similar semantics to the built-in `zip()` function in Python, with the main difference being that the `datasets` argument can be an arbitrary nested structure of `Dataset` objects. For example:

```
# NOTE: The following examples use `{ ... }` to represent the
# contents of a dataset.
a = { 1, 2, 3 }
b = { 4, 5, 6 }
c = { (7, 8), (9, 10), (11, 12) }
d = { 13, 14 }

# The nested structure of the `datasets` argument determines the
# structure of elements in the resulting dataset.
Dataset.zip((a, b)) == { (1, 4), (2, 5), (3, 6) }
Dataset.zip((b, a)) == { (4, 1), (5, 2), (6, 3) }

# The `datasets` argument may contain an arbitrary number of
# datasets.
Dataset.zip((a, b, c)) == { (1, 4, (7, 8)),
                            (2, 5, (9, 10)),
                            (3, 6, (11, 12)) }

# The number of elements in the resulting dataset is the same as
# the size of the smallest dataset in `datasets`.
Dataset.zip((a, d)) == { (1, 13), (2, 14) }
```

Args:

- `datasets`: A nested structure of datasets.

Returns:

A `Dataset`.

Last updated November 2, 2017.