"""

This module implements custom tf.data.datasets for twml.

"""

import numbers

from absl import logging

from kazoo.client import KazooClient

from libtwml import OPLIB

import tensorflow.compat.v1 as tf

from twml.constants import DEFAULT\_ZOOKEEPER\_BASE\_ZNODE, DEFAULT\_ZOOKEEPER\_HOST

class BlockFormatDataset(tf.data.Dataset):

"""A ``tf.data.Dataset`` comprising records from one or more TFRecord files."""

def \_\_init\_\_(self, filenames, compression\_type="auto", buffer\_size=1 << 20):

"""

Creates a ``BlockFormatDataset``.

Args:

filenames:

A `tf.string` tensor containing one or more filenames.

compression\_type:

A string specifying the compression type.

Can be one of 'gz' (or 'gzip'), 'none', 'auto' (default).

When compression\_type == 'auto', it is inferred from file extension.

buffer\_size:

Buffer size to be used during decompression. default: 1<<20.

"""

self.\_filenames = tf.convert\_to\_tensor(filenames, dtype=tf.string, name="filenames")

self.\_compression\_type = tf.convert\_to\_tensor(compression\_type.lower(), name="compression\_type")

self.\_buffer\_size = tf.convert\_to\_tensor(buffer\_size, dtype=tf.int64, name="buffer\_size")

# Parent class calss self.\_as\_variant\_tensor in init. So call this at the end.

super(BlockFormatDataset, self).\_\_init\_\_()

def \_as\_variant\_tensor(self):

"""

Create the resource handle for the dataset.

"""

try:

block\_format\_dataset = \_\_import\_\_("libtwml\_internal").OPLIB.block\_format\_dataset

return block\_format\_dataset(self.\_filenames)

except ImportError:

block\_format\_dataset = OPLIB.block\_format\_dataset\_v2

return block\_format\_dataset(self.\_filenames, self.\_compression\_type, self.\_buffer\_size)

def \_inputs(self):

return []

@property

def output\_shapes(self):

"""Return output shapes"""

return tf.TensorShape([])

@property

def output\_types(self):

"""Return output types"""

return tf.string

@property

def output\_classes(self):

"""Return output classes"""

return tf.Tensor

def downsample\_dataset(dataset, sample\_rate, rate\_name):

"""

Downsample a tf.data.Dataset at sample\_rate

"""

if sample\_rate is None or sample\_rate == 1.0:

return dataset

elif not isinstance(sample\_rate, numbers.Real):

raise TypeError("dataset %s must be a real number" % rate\_name)

elif sample\_rate <= 0 or sample\_rate > 1:

raise ValueError("dataset %s must be in range (0, 1])" % rate\_name)

return dataset.filter(lambda \_: tf.squeeze(tf.random\_uniform([1])) < sample\_rate)

def \_filenames\_dataset(files, shards=None, shard\_index=None):

"""

Get a tf.data.Dataset with file names from a list of files

Optionally shard the file list (see stream\_block\_format\_dataset)

"""

files = tf.data.Dataset.from\_tensor\_slices(files)

if [shards, shard\_index] != [None, None]:

logging.info("Sharding files dataset (index: %d, shards: %d)" % (shard\_index, shards))

files = files.shard(num\_shards=shards, index=shard\_index)

return files

def stream\_block\_format\_dataset(

files, parse\_fn, batch\_size, num\_threads,

shuffle=True, repeat=False,

block\_length=None, part\_file\_parallelism=None, file\_shuffle\_size=None,

record\_shuffle\_size=None, dataset\_fn=None,

keep\_rate=None, parts\_downsampling\_rate=None, prefetch\_size=2,

shards=None, shard\_index=None, shuffle\_files=True, interleave=True):

"""

Helper function to stream a list of part files.

Args:

files:

List of input files which will create a dataset.

parse\_fn:

A function that takes a byte tensor containing a datarecord and decodes it.

batch\_size:

The batch size for each step.

num\_threads:

Number of threads working on the data in parallel.

shuffle:

Shuffle records within each file using ``record\_shuffle\_size``. Defaults to True.

repeat:

Repeat the dataset indefinitely. Defaults to False.

Useful when you want to use an ``[train,eval]\_steps`` greater than the size of the dataset

(otherwise ``Estimator.[train,evaluate]`` stop when the end of the dataset is reached).

block\_length (optional):

Number of consecutive records to pull from a single part file.

Defaults to batch\_size.

part\_file\_parallelism (optional):

Number of part files to read from in parallel. Once a part file is completely read, it will

be replaced by the next part file in the part file list.

``num\_threads`` specifies a reader thread pool size, while ``part\_file\_parallelism`` specifies

the number of files to read from in parallel. If ``part\_file\_parallelism`` is greater than or

equal to ``num\_threads``, the reads will be distributed over ``num\_threads``. On the other hand,

if ``part\_file\_parallelism`` is smaller than``num\_threads``, it is very likely that the reader

thread pool will be underutilized, since it can never be the case that every reader thread has

a part file to read from.

file\_shuffle\_size (optional):

the buffer\_size used for shuffling of the list of files.

Defaults to 1000. For example, if you have 2000 files, the first

1000 files are shuffled together, iterated through, then the next 1000 files are shuffled

and iterated through.

record\_shuffle\_size (optional):

the ``buffer\_size`` used for shuffling records in each thread.

Defaults to ``batch\_size \* 8`` records.

dataset\_fn (optional):

A function of that modifies the dataset after it reads different interleaved parts files.

Defaults to:

.. code-block:: python

def dataset\_fn(dataset, parse\_fn, batch\_size):

return dataset.batch(batch\_size).map(parse\_fn, 1)

keep\_rate (optional):

A float value in (0.0, 1.0] that indicates to drop records according to the Bernoulli

distribution with p = 1 - keep\_rate.

Defaults to None (no records dropped).

parts\_downsampling\_rate (optional):

A float value in ``(0.0, 1.0]`` that indicates the factor by which to downsample part files.

For example, a value of 0.2 means only 20 percent of part files become part of the dataset.

Note that this argument is only useful in conjunction with a [train,eval]\_steps of -1

(that is, when the entire dataset is used). Furthermore, note that even in this case, each

epoch will see a different set of part files. This is because new part files are re-sampled

every epoch. In other words, this argument is only provided for backwards compatibility with

DeepBird v1. We recommend you use a smaller [train,eval]\_steps (or specify a keep\_rate)

instead.

shards (optional):

Number of partitions to shard the dataset into. This is useful for codistillation and other

techniques that require each worker to train on disjoint partitions of the dataset.

The dataset is not sharded by default.

shard\_index (optional):

Which partition of the dataset to use if ``shards`` is set.

shuffle\_files (optional):

Shuffle the list of files. Defaults to True.

When False, files are iterated in the order they are passed in.

interleave (optional):

Interleave records from multiple files in parallel. Defaults to True.

Returns:

tf.data.DataSet of batches of HashedDataRecord resource handles decoded and streamed online.

"""

# Creating a dataset from an input directory

files = \_filenames\_dataset(files, shards=shards, shard\_index=shard\_index)

file\_shuffle\_size = file\_shuffle\_size if file\_shuffle\_size is not None else 100000

record\_shuffle\_size = record\_shuffle\_size if record\_shuffle\_size is not None else (batch\_size \* 8)

block\_length = block\_length if block\_length is not None else batch\_size

logging.info("NUM\_THREADS: %d", num\_threads)

if repeat:

files = files.repeat()

if shuffle\_files:

# Randomly shuffle the files list.

files = files.shuffle(buffer\_size=file\_shuffle\_size)

# Downsample parts files

files = downsample\_dataset(files, parts\_downsampling\_rate, "parts\_downsampling\_rate")

# Interleave the result from BlockFormatDataset

# block\_length == batch\_size results in batch\_size records being read from a single file.

def map\_fn(filenames):

'''function that maps each filename to a BlockFormatDataset'''

# reach each file using BlockFormatDataset

dataset = BlockFormatDataset(filenames)

# early prefetching can sometimes improve performance (like on GCS)

dataset = dataset.prefetch(tf.data.experimental.AUTOTUNE)

# Shuffling before repeating ensures strong ordering.

if shuffle:

dataset = dataset.shuffle(buffer\_size=record\_shuffle\_size)

return dataset

if interleave:

part\_file\_parallelism = num\_threads if part\_file\_parallelism is None else part\_file\_parallelism

dataset = files.interleave(

map\_fn, cycle\_length=part\_file\_parallelism, block\_length=block\_length, num\_parallel\_calls=num\_threads)

else:

dataset = files.flat\_map(map\_fn)

# Downsample DataRecords

dataset = downsample\_dataset(dataset, keep\_rate, "keep\_rate")

if dataset\_fn is None:

# Create a batch of datarecords and decode them

return dataset.batch(batch\_size).map(parse\_fn, num\_parallel\_calls=tf.data.experimental.AUTOTUNE).prefetch(prefetch\_size)

return dataset\_fn(dataset, parse\_fn, batch\_size)

def cx\_zk\_path(path):

if path is None:

raise ValueError("Path for zookeeper dataset pointer is None. You must specify a path.")

return\_path = "/".join([DEFAULT\_ZOOKEEPER\_BASE\_ZNODE, path])

logging.info("Zookeeper path is: {}".format(return\_path))

return return\_path

def zookeeper\_ordered\_dataset(

files, parse\_fn, batch\_size, zk\_counter\_path, repeat=False,

num\_threads=2, block\_length=None, part\_file\_parallelism=None,

batch\_shuffle\_size=None, file\_keep\_rate=None, record\_keep\_rate=None,

prefetch\_size=2, interleave=False, dataset\_fn=None, verbose=False):

"""

Make a tf.Dataset given an ordered list of filenames, using Zookeeper to keep track of

which file to read, and to coordinate multiple workers.

Args:

files:

ordered list of (typically HDFS) filenames. This must remain consistent

between different workers, and between worker restarts (e.g. in the case

of instance failure or preemption).

To ensure this remains consistent, consider using the --train.files\_list

option from DataRecordTrainer.

parse\_fn:

A function that takes a byte tensor containing a datarecord and decodes it.

batch\_size:

The batch size for each step.

zk\_counter\_path:

Path under the root node for the underlying zookeeper shared counter that

is used to coordinate distributed iteration over the list of files.

Full path will be `'/'.join([DEFAULT\_ZOOKEEPER\_BASE\_ZNODE, zk\_counter\_path])`.

repeat:

Default False. Set True to repeat over the files forever.

num\_threads:

Default 2. Number of threads working on the data in parallel.

Only used if interleave=True.

block\_length:

Default None. Number of consecutive records to pull from a single part file.

If None, then block\_length=batch\_size will be used.

Only used if interleave=True.

part\_file\_parallelism:

Default None. Number of part files to read from in parallel. Once a part file is completely

read, it will be replaced by the next part file indicated by the zookeeper counter.

Only used if interleave=True.

``num\_threads`` specifies a reader thread pool size, while ``part\_file\_parallelism`` specifies

the number of files to read from in parallel. If ``part\_file\_parallelism`` is greater than or

equal to ``num\_threads``, the reads will be distributed over ``num\_threads``. On the other hand,

if ``part\_file\_parallelism`` is smaller than``num\_threads``, it is very likely that the reader

thread pool will be underutilized, since it can never be the case that every reader thread has

a part file to read from.

batch\_shuffle\_size:

Default None. Size of shuffle buffer, for shuffling that will be applied after batching.

if None, then batches will not be shuffled. Ignored if dataset\_fn is provided.

file\_keep\_rate:

Default None. Fraction of files to keep, or None to keep all files.

record\_keep\_rate:

Default None. Fraction of records to keep, or None to keep all records.

prefetch\_size:

Default 2. Number of parsed batches to prefetch. Ignored if dataset\_fn is provided.

interleave:

Default False. Set True to use tf.data.Dataset.interleave rather than flat\_map.

dataset\_fn:

A function that is applied to the dataset of individual records, after

these have been read from the parts files.

If ``None`` (the default), the behavior will be as though dataset\_fn were set to:

.. code-block:: python

def dataset\_fn(dataset, parse\_fn, batch\_size):

dataset = dataset.batch(batch\_size)

dataset = dataset.map(parse\_fn, tf.data.experimental.AUTOTUNE)

if batch\_shuffle\_size:

dataset = dataset.shuffle(batch\_shuffle\_size)

return dataset.prefetch(prefetch\_size)

verbose:

Default False. Set True to log the names of files loaded by TF.

"""

block\_length = batch\_size if block\_length is None else block\_length

part\_file\_parallelism = num\_threads if part\_file\_parallelism is None else part\_file\_parallelism

def zk\_index\_generator(my\_files=files):

zk = KazooClient(hosts=DEFAULT\_ZOOKEEPER\_HOST)

zk.start()

my\_counter = zk.Counter(cx\_zk\_path(zk\_counter\_path), default=0)

while True:

my\_counter += 1

counter\_pre\_value = my\_counter.pre\_value

if repeat:

counter\_pre\_value = counter\_pre\_value % len(my\_files)

if counter\_pre\_value >= len(my\_files):

break

else:

chosen\_file = my\_files[counter\_pre\_value]

if verbose:

logging.info("{}. yielding {}".format(counter\_pre\_value, chosen\_file))

yield chosen\_file

zk.stop()

files = tf.data.Dataset.from\_generator(zk\_index\_generator, tf.string)

# Downsample parts files

files = downsample\_dataset(files, file\_keep\_rate, "file\_keep\_rate")

def map\_fn(filenames):

return BlockFormatDataset(filenames).prefetch(20)

# Dont interleave for sequential training

if interleave:

dataset = files.interleave(

map\_fn,

cycle\_length=part\_file\_parallelism,

block\_length=block\_length,

num\_parallel\_calls=num\_threads)

else:

dataset = files.flat\_map(map\_fn)

# Downsample DataRecords

dataset = downsample\_dataset(dataset, record\_keep\_rate, "record\_keep\_rate")

if dataset\_fn is None:

# Create a batch of datarecords and decode them

dataset = dataset.batch(batch\_size)

dataset = dataset.map(parse\_fn, num\_parallel\_calls=tf.data.experimental.AUTOTUNE)

# shuffle after batching and parsing for performance reasons

# faster b/c 1 random selection is made per batch rather than per record

if batch\_shuffle\_size:

dataset = dataset.shuffle(buffer\_size=batch\_shuffle\_size)

dataset = dataset.prefetch(prefetch\_size)

else:

dataset = dataset\_fn(dataset, parse\_fn, batch\_size)

return dataset