tftables Documentation

Release 1.0.0

ghcollin

Feb 15, 2018

Contents

tftables allows convenient access to HDF5 files with Tensorflow. A class for reading batches of data out of arrays or tables is provided. A secondary class wraps both the primary reader and a Tensorflow FIFOQueue for straight-forward streaming of data from HDF5 files into Tensorflow operations.

The library is backed by multitables for high-speed reading of HDF5 datasets. `multitables` is based on PyTables (`tables`), so this library can make use of any compression algorithms that PyTables supports.

CHAPTER 1

Contents

1.1 Quick Start

1.1.1 Installation

```
pip install tftables
```

Alternatively, to install from HEAD, run

```
pip install git+https://github.com/ghcollin/tftables.git
```

You can also download or clone the repository and run

```
python setup.py install
```

`tftables` depends on `multitables`, `numpy` and `tensorflow`. The package is compatible with the latest versions of python 2 and 3.

1.1.2 Quick start

An example of accessing a table in a HDF5 file.

```
import tftables
import tensorflow as tf

with tf.device('/cpu:0'):
    # This function preprocesses the batches before they
    # are loaded into the internal queue.
    # You can cast data, or do one-hot transforms.
    # If the dataset is a table, this function is required.
    def input_transform(tbl_batch):
        labels = tbl_batch['label']
```

```
        data = tbl_batch['data']

        truth = tf.to_float(tf.one_hot(labels, num_labels, 1, 0))
        data_float = tf.to_float(data)

        return truth, data_float

    # Open the HDF5 file and create a loader for a dataset.
    # The batch_size defines the length (in the outer dimension)
    # of the elements (batches) returned by the reader.
    # Takes a function as input that pre-processes the data.
    loader = tftables.load_dataset(filename=self.test_filename,
                                   dataset_path=self.test_mock_data_path,
                                   input_transform=input_transform,
                                   batch_size=20)

# To get the data, we dequeue it from the loader.
# Tensorflow tensors are returned in the same order as input_transformation
truth_batch, data_batch = loader.dequeue()

# The placeholder can then be used in your network
result = my_network(truth_batch, data_batch)

with tf.Session() as sess:

    # This context manager starts and stops the internal threads and
    # processes used to read the data from disk and store it in the queue.
    with loader.begin(sess):
        for _ in range(num_iterations):
            sess.run(result)
```

If the dataset is an array instead of a table. Then `input_transform` can be omitted if no pre-processing is required. If only a single pass through the dataset is desired, then you should pass `cyclic=False` to `load_dataset`.

1.1.3 Examples

See the *How-To* for more in-depth documentation, and the unit tests for complete examples.

1.2 How To

Use of the library starts with creating a `TableReader` object.

```
import tftables
reader = tftables.open_file(filename="/path/to/h5/file", batch_size=10)
```

Here the batch size is specified as an argument to the `open_file` function. The batch_size defines the length (in the outer dimension) of the elements (batches) returned by `reader`.

1.2.1 Accessing a single array

Suppose you only want to read a single array from your HDF5 file. Doing this is quite straight-forward. Start by getting a tensorflow placeholder for your batch from `reader`.

```
array_batch_placeholder = reader.get_batch(
    path = '/h5/path',  # This is the path to your array inside the HDF5 file.
    cyclic = True,      # In cyclic access, when the reader gets to the end of the
                        # array, it will wrap back to the beginning and continue.
    ordered = False     # The reader will not require the rows of the array to be
                        # returned in the same order as on disk.
)

# You can transform the batch however you like now.
# For example, casting it to floats.
array_batch_float = tf.to_float(array_batch_placeholder)

# The data can now be fed into your network
result = my_network(array_batch_float)

with tf.Session() as sess:
    # The feed method provides a generator that returns
    # feed_dict's containing batches from your HDF5 file.
    for i, feed_dict in enumerate(reader.feed()):
        sess.run(result, feed_dict=feed_dict)
        if i >= N:
            break

# Finally, the reader should be closed.
reader.close()
```

Note that be default, the `ordered` argument to `get_batch` is set to `True`. If you require the rows of the array to be returned in the same order as they are on disk, then you should leave it as `ordered = True`. However, this may result in a performance penalty. In machine learning, rows of a dataset often represent independent examples, or data points. Thus their ordering is not important.

1.2.2 Accessing a single table

When reading from a table, the `get_batch` method returns a dictionary. The columns of the table form the keys of this dictionary, and the values are tensorflow placeholders for batches of each column. If one of the columns has a compound datatype, then its corresponding value in the dictionary will itself be a dictionary. In this way, recursive compound datatypes will give recursive dictionaries.

For example, if your table just had two columns, named `label` and `data`, then you could use:

```
table_batch = reader.get_batch(
    path = '/path/to/table',
    cyclic = True,
    ordered = False
)

label_batch = table_batch['label']
data_batch = table_batch['data']
```

If your table was a bit more complicated, with columns named `label` and `value`. And the `value` column has a compound type with fields named `image` and `lidar`, then you could use:

```
table_batch = reader.get_batch(
    path = '/path/to/complex_table',
    cyclic = True,
    ordered = False
```

```
)

label_batch = table_batch['label']
value_batch = table_batch['value']

image_batch = value_batch['image']
lidar_batch = value_batch['lidar']
```

1.2.3 Using a FIFO queue

Copying data to the GPU through a `feed_dict` is notoriously slow in Tensorflow. It is much faster to buffer data in a queue. You are free to manage your own queues, but a helper class is included to make this task easier.

```
# As before
array_batch_placeholder = reader.get_batch(
    path = '/h5/path',
    cyclic = True,
    ordered = False)
array_batch_float = tf.to_float(array_batch_placeholder)

# Now we create a FIFO Loader
loader = reader.get_fifoloader(
    queue_size = 10,              # The maximum number of elements that the
                                  # internal Tensorflow queue should hold.
    inputs = [array_batch_float], # A list of tensors that will be stored
                                  # in the queue.
    threads = 1                   # The number of threads used to stuff the
                                  # queue. If ordered access to a dataset
                                  # was requested, then only 1 thread
                                  # should be used.
)

# Batches can now be dequeued from the loader for use in your network.
array_batch_cpu = loader.dequeue()
result = my_network(array_batch_cpu)

with tf.Session() as sess:

    # The loader needs to be started with your Tensorflow session.
    loader.start(sess)

    for i in range(N):
        # You can now cleanly evaluate your network without a feed_dict.
        sess.run(result)

    # It also needs to be stopped for clean shutdown.
    loader.stop(sess)

# Finally, the reader should be closed.
reader.close()
```

Non-cyclic access

If you are classifying a dataset, rather than training a model, then you probably only want to run through the dataset once. This can be done by passing `cyclic = False` to `get_batch`. Once finished, the internal Tensorflow queue

will throw an instance of the `tensorflow.errors.OutOfRangeError` exception to signal termination of the loop.

This can be caught manually with a try-catch block:

```
with tf.Session() as sess:
    loader.start(sess)

    try:
        # Keep iterating until the exception breaks the loop
        while True:
            sess.run(result)
    # Now silently catch the exception.
    except tf.errors.OutOfRangeError:
        pass

    loader.stop(sess)
```

A slightly more elegant solution is to use a context manager supplied by the loader class:

```
with tf.Session() as sess:
    loader.start(sess)

    # This context manager suppresses the exception.
    with loader.catch_termination():
        # Keep iterating until the exception breaks the loop
        while True:
            sess.run(result)

    loader.stop(sess)
```

Start stop context manager

In either cyclic or non-cyclic access, we can use a context manager to start and stop the loader class.

```
with tf.Session() as sess:
    with loader.begin(sess):
        # Loop
```

1.2.4 Quick access to a single dataset

It is highly recommended that you use a single dataset, this allows you to use unordered access which is a fastest way of reading data. If you have multiple sources of data, such as labels and images, then you should organise them into a table. This also has performance benefits due to the locality of the data.

When you only have one dataset, the function `load_dataset` is provided to set up the reader and loader for you. Any preprocessing that need to be done CPU side before loading into the queue can be written as a function that generates a Tensorflow graph. This input transformation function is fed into `load_dataset` as an argument.

The input transform function should return a list of tensors that will be stored in the queue. The input transform is required when the dataset is a table, as the dictionary needs to be turned into a list.

```
# This function preprocesses the batches before they
# are loaded into the internal queue.
# You can cast data, or do one-hot transforms.
# If the dataset is a table, this function is required.
```

```
def input_transform(tbl_batch):
    labels = tbl_batch['label']
    data = tbl_batch['data']

    truth = tf.to_float(tf.one_hot(labels, num_labels, 1, 0))
    data_float = tf.to_float(data)

    return truth, data_float

# Open the HDF5 file and create a loader for a dataset.
# The batch_size defines the length (in the outer dimension)
# of the elements (batches) returned by the reader.
# Takes a function as input that pre-processes the data.
loader = tftables.load_dataset(filename='path/to/h5_file.h5',
                               dataset_path='/internal/h5/path',
                               input_transform=input_transform,
                               batch_size=20)

# To get the data, we dequeue it from the loader.
# Tensorflow tensors are returned in the same order as input_transformation
truth_batch, data_batch = loader.dequeue()

# The placeholder can then be used in your network
result = my_network(truth_batch, data_batch)

with tf.Session() as sess:

    # This context manager starts and stops the internal threads and
    # processes used to read the data from disk and store it in the queue.
    with loader.begin(sess):
        for _ in range(num_iterations):
            sess.run(result)
```

When using `load_dataset` the reader is automatically closed when the loader is stopped.

1.2.5 Accessing multiple datasets

If your HDF5 file has multiple datasets (multiple arrays, tables or both) then you should write a script to transform it into a file with only a single table. If this isn't possible, then you can access the datasets directly through `tftables`, but must do so using ordered access (otherwise the datasets can get out of sync).

```
# Use get_batch to access the table.
# Both datasets must be accessed in ordered mode.
table_batch_dict = reader.get_batch(
    path = '/internal/h5_path/to/table',
    ordered = True)
col_A_pl, col_B_pl = table_batch_dict['col_A'], table_batch_dict['col_B']

# Now use get_batch again to access an array.
# Both datasets must be accessed in ordered mode.
labels_batch = reader.get_batch('/my_label_array', ordered = True)
truth_batch = tf.one_hot(labels_batch, 2, 1, 0)

# The loader takes a list of tensors to be stored in the queue.
# When accessing in ordered mode, threads should be set to 1.
loader = reader.get_fifoloader(
```

```
    queue_size = 10,
    inputs = [truth_batch, col_A_pl, col_B_pl],
    threads = 1)

# Batches are taken out of the queue using a dequeue operation.
# Tensors are returned in the order they were given when creating the loader.
truth_cpu, col_A_cpu, col_B_cpu = loader.dequeue()

# The dequeued data can then be used in your network.
result = my_network(truth_cpu, col_A_cpu, col_B_cpu)

with tf.Session() as sess:
    with loader.begin(sess):
        for _ in range(N):
            sess.run(result)

reader.close()
```

Ordered access is enabled be default when using `get_batch` as a safety measure. It is disabled when using `load_dataset` as that function restricts access to a single dataset.

1.3 Reference

`tftables.`**`open_file`**(*filename*, *batch_size*, ***kw_args*)

Open a HDF5 file for streaming with multitables. Batches will be retrieved with size `batch_size`. Additional keyword arguments will be passed to the `multitables.Streamer` object.

Parameters

- **`filename`** – Filename for the HDF5 file to be read.
- **`batch_size`** – The size of the batches to be fetched by this reader.
- **`kw_args`** – Optional arguments to pass to multitables.

Returns A FileReader instance.

`tftables.`**`load_dataset`**(*filename*, *dataset_path*, *batch_size*, *queue_size=8*, *input_transform=None*, *ordered=False*, *cyclic=True*, *processes=None*, *threads=None*)

Convenience function to quickly and easily load a dataset using best guess defaults. If a table is loaded, then the `input_transformation` argument is required. Returns an instance of `FIFOQueueLoader` that loads this dataset into a fifo queue.

This function takes a single argument, which is either a tensorflow placeholder for the requested array or a dictionary of tensorflow placeholders for the columns in the requested table. The output of this function should be either a single tensorflow tensor, a tuple of tensorflow tensors, or a list of tensorflow tensors. A subsequent call to `loader.dequeue()` will return tensors in the same order as `input_transform`.

For example, if an array is stored in uint8 format, but we want to cast it to float32 format to do work on the GPU, the `input_transform` would be:

```
def input_transform(ary_batch):
    return tf.cast(ary_batch, tf.float32)
```

If, instead we were loading a table with column names `label` and `data` we need to transform this into a list. We might use something like the following to also do the one hot transform.

```
def input_transform(tbl_batch):
    labels = tbl_batch['labels']
    data = tbl_batch['data']

    truth = tf.to_float(tf.one_hot(labels, num_labels, 1, 0))
    data_float = tf.to_float(data)

    return truth, data_float
```

Then the subsequent call to `loader.dequeue()` returns these int the same order:

```
truth_batch, data_batch = loader.dequeue()
```

By default, this function does not preserve on-disk ordering, and gives cyclic access. The disk ordering can be preserved using the `ordered` argument; however, this may result in slower read performance.

Parameters

- **filename** – The filename to the HDF5 file.
- **dataset_path** – The internal HDF5 path to the dataset.
- **batch_size** – The size of the batches to be loaded into tensorflow.
- **queue_size** – The size of the tensorflow FIFO queue.
- **input_transform** – A function that transforms the batch before being loaded into the queue.
- **ordered** – Preserve the on-disk ordering of the requested dataset.
- **cyclic** – Data will be loaded in an endless loop that wraps around the end of the dataset.
- **processes** – Number of concurrent processes that multitables should use to read data from disk.
- **threads** – Number of threads to use to preprocess data and load the FIFO queue.

Returns a loader for the dataset

class tftables.**FileReader**(*filename*, *batch_size*, ***kw_args*)

This class reads batches from datasets in a HDF5 file.

close()

Closes the internal queue, signaling the background processes to stop. This calls the multitables.Streamer.Queue.close method.

Returns None

feed()

Generator for feeding a tensorflow operation. Each iteration returns a feed_dict that contains the data for one batch. This method reads data for *all* placeholders created.

Returns A generator which yields tensorflow feed_dicts

get_batch(*path*, ***kw_args*)

Get a Tensorflow placeholder for a batch that will be read from the dataset located at path. Additional key word arguments will be forwarded to the get_queue method in multitables. This defaults the multitables arguments *cyclic* and *ordered* to true.

When ordering of batches is unimportant, the *ordered* argument can be set to False for potentially better performance. When reading from multiple datasets (eg; when examples and labels are in two different arrays), it is recommended to set *ordered* to True to preserve synchronisation.

If the dataset is a table (or other compound-type array) then a dictionary of placeholders will be returned instead. The keys of this dictionary correspond to the column names of the table (or compound sub-types).

Parameters

- **path** – The internal HDF5 path to the dataset to be read.
- **kw_args** – Optional arguments to be forwarded to multitables.

Returns Either a placeholder or a dictionary depending on the type of dataset. If the dataset is a plain array, a placeholder representing once batch is returned. If the dataset is a table or compound type, a dictionary of placeholders is returned.

get_fifoloader(*queue_size*, *inputs*, *threads=None*)

Convenience method for creating a FIFOQueueLoader object. See the FIFOQueueLoader constructor for documentation on parameters.

Parameters

- **queue_size** –
- **inputs** –
- **threads** – Defaults to 1 if ordered access to this reader was requested, otherwise defaults to 2.

Returns

class tftables.**FIFOQueueLoader**(*reader*, *size*, *inputs*, *threads=1*)

A class to handle the creation and population of a Tensorflow FIFOQueue.

begin(**args*, ***kwds*)

Convenience context manager for starting and stopping the loader. :param tf_session: The current Tensorflow session. :param catch_termination: Catch the termination of the loop for non-cyclic access. :return:

static catch_termination()

In non-cyclic access, once the end of the dataset is reached, an exception is called to halt all access to the queue. This context manager catches this exception for silent handling of the termination condition. :return:

dequeue()

Returns a dequeue operation. Elements defined by the input tensors and supplied by the reader are returned from this operation. This calls the dequeue method on the internal Tensorflow FIFOQueue.

Returns A dequeue operation.

start(*sess*)

Starts the background threads. The enqueue operations are run in the given Tensorflow session.

Parameters **sess** – Tensorflow session.

Returns None

stop(*sess*)

Stops the background threads, and joins them. This should be called after all operations are complete.

Parameters **sess** – The Tensorflow operation that this queue loader was started with.

Returns

CHAPTER 2

Licence

This software is distributed under the MIT licence. See the LICENSE.txt file for details.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

t

Index