tf_ops Documentation

Release 0.0.1

Fergal Cotter

Dec 12, 2018

Contents:

CHAPTER 1

Dataset Loading

This repo is aimed at being a centralized resource for loading in commonly used image datasets like CIFAR, PASCAL VOC, MNIST, ImageNet and others.

Some of these datasets will fit easily on disk (CIFAR and MNIST), but many of the others won't. This means we have to set up threads to load them as we need them into memory. Tensorflow provides some ability to do this, but after several attempts at using these resources, we found them far too opaque and difficult to use. This package does essentially the same thing as what tensorflow does, but using python's threading, multiprocessing and queue packages.

1.1 Threads vs Processes

Initially this package would only use Python's threading package to parallelize tasks. It quickly became apparent that this caps the benefits of parallelization, as all of these threads will only take up to 1 processor core. In reality, we want to be able to take up more processors for data loading to reduce bottlenecks. It is still untested, but we are adding in multiprocess support for the heavy lifting tasks (in particular, loading and preprocessing images into *The ImageQueue*).

1.2 Dataset Specific Usage

For instructions on how to call the functions to get images in for common datasets, see their help pages. These functions wrap around the *General Usage* functions and are provided for convenience. If your application doesn't quite fit into these functions, or if you have a new dataset, have a look at *General Usage*, as it was designed to make queueing for any dataset type as easy as possible.

- MNIST usage instructions
- CIFAR10/CIFAR100 usage instructions

1.3 General Usage

For the bigger datasets, we need 2 queues and several threads (perhaps on multiple processors) to load images in parallel.

1. A File Queue to store a list of file names. Sequencing can be done by shuffling the file names before inserting into the queue.
    - One thread should be enough to manage this queue.
2. An Image Queue to load images into.
    - Several threads will likely be needed to read file names from the file queue, load from disk, and put into the Image Queue. We may get away with running these all in one Python process, but may need to use more.

1.3.1 The FileQueue

A FileQueue is used to store a list of file names (e.g. jpegs). This is also the location of sequencing (there is an option to shuffle the entries in this queue when adding) and where we set the limits on the number of epochs processed (if we wish to). For example, this would set up a file queue for 50 epochs:

```
import dataset_loading as dl
IM_DIR = /path/to/images
files = os.listdir(IM_DIR)
files = [f for f in files if os.path.splitext(f)[1] == '.jpeg']
file_queue = dl.FileQueue()
file_queue.load_epochs(files, max_epochs=50)
...
...
file_queue.join()
```

The *load_epochs* method will also start a single thread to manage the queue and refill it if it's getting low (shuffling along as it goes).

If you know what the labels are, you should also feed them to the File Queue alongside the file names in a list of (file, label) tuples. E.g.:

```
# Assume <labels> is a list of all of the labels and <files> is a
# list of the files.
file_queue = dl.FileQueue()
file_queue.load_epochs(list(zip(files, labels)), max_epochs=float('inf'))
```

Note that when you are done with the queue, you should call the queue's *join* method, which will make sure the queue is empty and the loader thread exits.

1.3.2 The ImageQueue

An ImageQueue to hold a set amount of images (not the entire batch, but enough to keep the main program happily fed). This class has a method we call for starting image reader threads (again, you can choose how many of these you need to meet your main's demand). Following the above code, you add an image queue like so:

```
img_queue = dl.ImgQueue(maxsize=1000)
img_queue.start_loaders(file_queue, num_threads=3, img_dir=IM_DIR)
# Wait for the image queue to fill up
```

(continues on next page)

(continued from previous page)

```
sleep(2)
data, labels = img_queue.get_batch(batch_size=100)
...
...
img_queue.join()
```

The ImgQueue.start_loaders method will start *num_threads* threads, each of which read from the file_queue, load from disk and feed into the image queue.

If you want the loaders to pre-process images before putting them into the image queue, you can provide a callable to ImgQueue.start_loaders to do this (see its docstring for more info). For example:

```
img_queue = dl.ImgQueue()
def preprocess(x):
    x = x.astype(np.float32)
    x = x - np.mean(x)
    x = x/max(1, np.std(x))
    return x
img_queue.start_loaders(file_queue, num_threads=3, transform=preprocess)
```

The ImgQueue.get_batch method has two extra options (*block* and *timeout*), instructing it how to handle cases when the image queue doesn't have a full batch worth of images (should we return with whatever's there, or wait for the loaders to catch up?). See its docstring for more info.

For synchronization with epochs, the ImageQueue has an attribute *last_batch* that will be set to true when an epoch's worth of images have been pulled from the ImageQueue.

```
data, labels = img_queue.get_batch(batch_size=100)
last_batch = img_queue.last_batch
if last_batch:
    # Print summary info...
```

You can monitor the queue size and fetch times for the ImgQueue too (to check whether you need to tweak some settings). This works by printing out info to a tensorboard summary file (currently only supported way of doing it). All you need to do is create a *tf.summary.FileWriter* (you can use the same one the rest of your main program is using), and call the ImgQueue.add_logging method. This will add the data as a to your tensorboard file.

```
img_queue = dl.ImgQueue()
def preprocess(x):
    x = x.astype(np.float32)
    x = x - np.mean(x)
    x = x/max(1, np.std(x))
    return x
img_queue.start_loaders(file_queue, num_threads=3, transform=preprocess)
file_writer = tf.summary.FileWriter('./log', tf.get_default_graph())
# Write period is the sample period in numbers of batches for dumping data
img_queue.add_logging(file_writer, write_period=10)
```

Note that when you are done with the queue, you should call the queue's *join* method, which will make sure the queue is empty and the loader thread exits.

1.3.3 Small Datasets

If you have a special case where the dataset is small, and so can fit into memory (like CIFAR or MNIST), then you won't need the same complexity to get batches of data and labels. However, it may still be beneficial to use the ImgQueue class for two reasons:

- Keeps the same programmatic interface regardless of the dataset
- May still want to parallelize things if you want to do preprocessing of images before putting them in the queue.

For this, use ImgQueue.take_dataset instead of ImgQueue.start_loaders. This method also has options like whether to shuffle the samples or not (will shuffle by default), and can take a callable function to apply to the images before putting them in the queue. The default number of threads to create is 1, but this can be increased with the *num_threads* parameter.

Note: **to avoid duplicating things in memory, the ImgQueue will not copy the data/labels**. This means that once your main program calls the *take_dataset* method, it shouldn't modify the arrays.

E.g.

```
import dataset_loading as dl
import dataset_loading.cifar as dlcifar
train_d, train_l, test_d, test_l, val_d, val_l = \
    dlcifar.load_cifar_data('/path/to/data')
img_queue = dl.ImgQueue()
img_queue.take_dataset(train_d, train_l)
data, labels = img_queue.get_batch(100)
# Or say we want to use more parallel threads and morph the image
def preprocess(x):
    x = x.astype(np.float32)
    x = x - np.mean(x)
    x = x/max(1, np.std(x))
    return x
img_queue = dl.ImgQueue()
img_queue.take_dataset(train_d, train_l, num_threads=3,
                       transform=preprocess)
data, labels = img_queue.get_batch(100)
```

1.4 Installation

Direct install from github (useful if you use pip freeze). To get the master branch, try:

```
$ pip install -e git+https://github.com/fbcotter/dataset_loading#egg=dataset_loading
```

or for a specific tag (e.g. 0.0.1), try:

```
$ pip install -e git+https://github.com/fbcotter/dataset_loading.git@0.0.1
↪#egg=dataset_loading
```

Download and pip install from Git:

```
$ git clone https://github.com/fbcotter/dataset_loading
$ cd dataset_loading
$ pip install -r requirements.txt
$ pip install -e .
```

It is recommended to download and install (with the editable flag), as it is likely you'll want to tweak things/add functions more quickly than we can handle pull requests.

1.5 Further documentation

There is more documentation available online and you can build your own copy via the Sphinx documentation system:

```
$ python setup.py build_sphinx
```

Compiled documentation may be found in `build/docs/html/` (index.html will be the homepage)

CHAPTER 2

FileQueue

Typically, you will set up a file queue to give to an Image Loader thread and will never need to touch it, but if you do wish to use it directly, there are some things to note. For these notes, it is useful to look at the typical usage:

```
import dataset_loading as dl
files = [<some list of filenames or a list of tuples of (filenames, labels)>]
file_queue = dl.FileQueue()
file_queue.load_epochs(files, max_epochs=50)
```

Calling the load_epochs function actually spins up a thread to manage the file queue. This thread doesn't have to do much (so we only use 1), but it will refill the queue if it starts to get too low (<50% of one epoch). Initially, it will load 10 epochs worth of the <files> list into the FileQueue. This is not too important a quantity, we just want it to be big enough so that calls to the FileQueue.get() shouldn't be blocking most of the time, and not so big that the FileQueue takes up lots of memory.

In case you happen to request a lot of files when the queue is relatively empty, it would be a good idea to put a small timeout on the get(). Not so long (as you may have hit the end of the epoch limit and the queue will not refill!) and long enough to allow the FileQueue manager thread to detect the queue has emptied and give it time to fill up. Perhaps 10ms should work, i.e.:

```
list_of_files = [file_queue.get(timeout=0.01) for _ in range(1000)]
```

For more info on the FileQueue, see its `docstring`.

CHAPTER 3

ImageQueue

The Image Queue is the interface between the package and your main program. Once you have built a file queue to store the file names to read in, you can create an ImageQueue. Standard would look like this:

```
import dataset_loading as dl
file_queue = dl.FileQueue()
file_queue.load_epochs(<list_of_files>, max_epochs=50)
img_queue = dl.ImgQueue()
img_queue.start_loaders(file_queue, num_threads=3)
# Wait for the image queue to fill up
sleep(5)
img_queue.get_batch(<batch_size>)
```

Calling the start_loaders method spins up <num_threads> threads to pull from the file queue and write to the image queue. See the `ImgQueue.start_loaders` docstring for more info on the parameters you have here, but note that this is where you set:

- Path offsets for the files in the file queue (in case the files in the file queue weren't the absolute path of the images).
- The size of the image to resize to. By default (a parameter of None), no resizing will be done.
- Any pre-filtering operation to be done to the images (e.g. contrast normalization).

E.g.:

```
def norm_image(x):
    adjusted_stddev = max(np.std(x), 1.0/np.sqrt(x.size))
    return (x-np.mean(x))/adjusted_stddev
imsize = (224,224)
path_offset = '/scratch/share/pascal'
img_queue.start_loaders(file_queue, num_threads=3, img_size=imsize,
                        img_dir=path_offset, transform=norm_image)
```

For more info on the ImgQueue, see its `docstring`.

3.1 Note

By default the *get_batch* function does NOT block. I.e. if you call it, asking for 100 samples but only 50 are available, it will return with 50. If you do not want this, then you can set the parameter *block=True*. You may also consider setting the *timeout* parameter to a sensible value.

3.2 Queue Monitoring

You can take advantage of tensorflow's tensorboard and plot out some queue statistics too. The dataset_loading package is meant to be able to work without tensorflow, so attempting these methods may throw warnings and not work. Logging is automatically done when calls to the *get_batch* method are made.

```
img_queue.start_loaders(file_queue, num_threads=3, transform=preprocess)
file_writer = tf.summary.FileWriter('./log', tf.get_default_graph())
# Write period is the sample period in numbers of batches for dumping data
img_queue.add_logging(file_writer, write_period=10)
```

3.3 Properties

Here are some useful properties of the ImgQueue class that may help you in designing your program:

- last_batch : True if the previously read batch was the last in the epoch. Reading this value resets it to false.
- epoch_size : The number of images in the epoch. Interpreted from the File Queue. Cannot always determine this.
- read_count : How many images have been read in the current epoch
- image_shape : Inspects the queue and gets the shape of the images in it. Useful to check what the output shape from any preprocessing steps done beforehand were.
- label_shape : Inspects the queue and gets the shape of the labels in it.

3.4 A note on the Order of Images coming from the ImgQueue

Note that even if you do not shuffle the samples in the file queue, it is likely that the samples in the Image Queue will come out in a different order between two runs. This is because of the inherent random nature of multiple feeder threads pushing to the Image Queue at different rates. For example, consider the below code:

```
from dataset_loading import FileQueue, ImgQueue
import os

# Samples is a directory with about 100 images in it
files = os.listdir('samples')
files = [os.path.join('samples', f) for f in files]

# Make the filename the label
files = [(f,f) for f in files]
fq1 = FileQueue()
fq2 = FileQueue()
iq1 = ImgQueue()
```

(continues on next page)

(continued from previous page)

```
iq2 = ImgQueue()
fq1.load_epochs(files, shuffle=False)
fq2.load_epochs(files, shuffle=False)
iq1.start_loaders(fq1)
iq2.start_loaders(fq2)
data1, labels1 = iq1.get_batch(10)
data2, labels2 = iq2.get_batch(10)

# Print out the two, they will likely be different
print('List 1:\n{}'.format('\n'.join(labels1)))
print('List 2:\n{}'.format('\n'.join(labels2)))
```

If you really want the images to come out in the same order on subsequent runs, you'll need to restrict the number of loader threads to 1. I.e. replace `iq.start_loaders(fq)` with `iq.start_loaders(fq, num_threads=1)`.

CHAPTER 4

MNIST

As this is a very commonly used dataset, there is a utility function to help load it in:

This is identical to the CIFAR function. To check out usage instructions, have a look there. **Note that the return size for MNIST will be 28x28x1**.

In particular, there exists two functions in this module that may be of use. The first is *dataset_loading.mnist.load_mnist_data()*, which can be used to load in MNIST without queues. There is an argument for this function to request it to download MNIST if you haven't already got it.

The second is `dataset_loading.cifar.get_mnist_queues()` which will load MNIST and put it into some queues. Although MNIST is very small and can easily fit into memory, the benefit of this is parallel processing can be used to prescale the data before feeding it to your network. For more examples on how to do this, see the page explaining loading in the CIFAR data.

CHAPTER 5

CIFAR 10 & 100 Datasets

As this is a very commonly used dataset, the *dataset_loading.cifar* module has some helper functions for handling it.

5.1 No Queues

If you don't want to use queues (possible for CIFAR as it is quite small), but still want to make use of a utility function to load it in, you can use the function *dataset_loading.cifar.load_cifar_data()*:

```
from dataset_loading import cifar
CIFAR_DIR = '/path/to/saved/dataset'
# The load function will return a tuple
trainx, trainy, testx, testy, valx, valy = cifar.load_cifar_data(
    CIFAR_DIR, cifar10=True, val_size=0, one_hot=False, download=False)
```

5.1.1 Downloading

If you don't have the data, you can get the helper functions to download it for you before putting the data into queues. In this case, it will be downloaded into the data_dir specified.

```
from dataset_loading import cifar
train_queue, test_queue, val_queue = cifar.get_cifar_queues(
    '/path/to/cifar/data', cifar10=True, download=True)
```

5.2 Loading CIFAR in Queues

If you want to handle the CIFAR datasets with the Queues this package builds, you can call the *dataset_loading.cifar.get_cifar_queues()*. This calls the above *load_cifar_data()* function, so also has the ability to download the data.

The best way to understand this function is to see how it is used.

```
from dataset_loading import cifar
from time import sleep
train_queue, test_queue, val_queue = cifar.get_cifar_queues(
    '/path/to/cifar/data', cifar10=True)
sleep(1)
data, labels = train_queue.get_batch(100)
test, labels = test_queue.get_batch(100)
val, labels = val_queue.get_batch(100)
```

5.2.1 Preprocessing

What if we want to preprocess images by removing their mean before putting them into the queue? The benefit of this is that when your main function is ready for the next batch, it doesn't have to do any of this preprocessing.

```
from dataset_loading import cifar
import numpy as np
from time import sleep
def preprocess(x):
    x = x.astype(np.float32)
    x = x - np.mean(x)
    return x
train_queue, test_queue, val_queue = cifar.get_cifar_queues(
    '/path/to/cifar/data', transform=preprocess, cifar10=True)
sleep(1)
data, labels = train_queue.get_batch(100)
test, labels = test_queue.get_batch(100)
val, labels = val_queue.get_batch(100)
```

Ok, easy enough. What about if we wanted to do some preprocessing to the train set, but not to the validation and test? This is commonly done to 'augment' your dataset.

```
from dataset_loading import cifar
import numpy as np
from time import sleep
# this augmentation just adds noise to the train data
def preprocess(x):
    x = x.astype(np.float32)
    x = x + 10*np.random.rand(32,32,3)
    return x
transform = (preprocess, None, None)
train_queue, test_queue, val_queue = cifar.get_cifar_queues(
    '/path/to/cifar/data', transform=transform, cifar10=True)
sleep(1)
data, labels = train_queue.get_batch(100)
test, labels = test_queue.get_batch(100)
val, labels = val_queue.get_batch(100)
```

5.2.2 Epoch Management

One of the main annoyances with tensorflow was the difficulty of swapping between train and validation sets in the same main function. Say if you wanted to process one epoch of training data, then run some validation tests before getting a new epoch of data. You would have to keep track manually of how many images you'd read as if you tried to set an epoch limit to 1, and then restart the queues, you would run into all sorts of problems.

The ImgQueue in this package has a *last_batch* property that indicates whether this epoch was the last one or not, providing an easy indication for the main program to move onto the validation stage. **This flag will get reset if you read from it**. This allows you to do something like the following:

```
from dataset_loading import cifar
import numpy as np
from time import sleep
train_queue, test_queue, val_queue = cifar.get_cifar_queues(
    '/path/to/cifar/data', cifar10=True)
sleep(1)
while True:
    while not train_queue.last_batch:
        data, labels = train_queue.get_batch(100)
        # process the data

    # Do some validation testing then
    # loop back to beginning and get the next batch
```

You can also inspect how many images have been processed in the current epoch by looking at the ImgQueue.read_count property. This shouldn't be modified however, as then the file queues and the image queue will get out of sync.

You can put a limit on the epoch count too. When this limit is reached, a *FileQueueDepleted* exception will be raised:

```
from dataset_loading import cifar, FileQueueDepleted
import numpy as np
from time import sleep
train_queue, test_queue, val_queue = cifar.get_cifar_queues(
    '/path/to/cifar/data', cifar10=True, max_epochs=50)
try:
    while not train_queue.last_batch:
        data, labels = train_queue.get_batch(100)
        # process the data

    # Do some validation testing then
    # loop back to beginning and get the next batch
except FileQueueDepleted:
    # No need to do any join calls for the threads as these should already
    # have exited, and if they haven't, they're daemon threads so no
    # worries.
    print('All done')
```

5.2.3 Selecting Queues

If you only want to get the train queue or the train and validation queues say, you can do this by using the *get_queues* parameter. E.g.:

```
from dataset_loading import cifar, FileQueueDepleted
import numpy as np
from time import sleep
train_queue, test_queue, val_queue = cifar.get_cifar_queues(
    '/path/to/cifar/data', cifar10=True, get_queues=(True, False, True))
assert test_queue is None
```

5.2.4 Queue Monitoring

See the *Queue Monitoring* section in the ImgQueue help.

5.3 Miscellanea

If you plan on only using the Dataset Specific functions, you should still be aware of some of the useful properties of the ImgQueue's received from the loading function. See *Properties* for a description of these.

CHAPTER 6

API Guide

6.1 Core Functions

class dataset_loading.**FileQueue**(*maxsize=0*)

Bases: queue.Queue

A queue to hold filename strings

This queue is used to indicate what order of jpeg files should be read. It may also be a good idea to put the class label alongside the filename as a tuple, so the main program can get access to both of these at the same time.

Create the class, and then call the load_epochs() method to start a thread to manage the queue and refill it as it gets low.

The maxsize is not provided as an option as we want the queue to be able to take entire epochs and not be restricted on the upper limit by a maxsize. The data should be no problem as the queue entries are only integers.

epoch_count

The current epoch count

epoch_size

Gives the size of one epoch of data

filling

Returns true if the file queue is being filled

get(*block=True*, *timeout=None*)

Get a single item from the Image Queue

join()

Method to signal any threads that are filling this queue to stop.

Threads will clean themselves up if the epoch limit is reached, but in case you want to kill them manually before that, you can signal them to stop here.

Note: Overloads the queue join method which normally blocks until the queue has been emptied. This will return even if the queue has data in it.

killed
Returns true if the queue has been asked to die

load_epochs(*files*, *shuffle=True*, *max_epochs=inf*)
Starts a thread to load the file names into the file queue.

Parameters

- **files** (*list*) – Can either be a list of filename strings or a list of tuples of (filenames, labels)
- **shuffle** (*bool*) – Whether to shuffle the list before adding it to the queue.
- **max_epochs** (*int or infinity*) – Maximum number of epochs to allow before queue manager stops refilling the queue.

Notes

Even if shuffle input is set to false, that doesn't necessarily mean that all images in the image queue will be in the same order across epochs. For example, if thread A pulls the first image from the list and then thread B gets the second 1. Thread A takes slightly longer to read in the image than thread B, so it gets inserted into the Image Queue afterwards. Trying to synchronize across both queues could be done, but it would add unnecessary complications and overhead.

Raises ValueError - If the files queue was empty

class dataset_loading.**ImgQueue**(*maxsize=1000*, *name=''*)
Bases: queue.Queue

A queue to hold images

This queue can hold images which will be loaded from the main program. Multiple file reader threads can fill up this queue as needed to make sure demand is met.

Each entry in the image queue will then be either tuple of (data, label). If the data is loaded using a filename queue and image loader threads and a label is not provided, each queue item will still be a tuple, only the label will be None. If you don't want to return this label, then you can set the nolabel input to the start_loaders function.

To get a batch of samples from the ImageQueue, see the *get_batch()* method.

If you are lucky enough to have an entire dataset that fits easily into memory, you won't need to use multiple threads to start loading data. You may however want to keep the same interface. In this case, you can call the take_dataset function with the dataset and labels, and then call the *get_batch()* method in the same manner.

Parameters

- **maxsize** (*positive int*) – Maximum number of images to hold in the queue. Needs to not be 0 or else it will keep filling up until you run out of memory.
- **name** (*str*) – Queue name

Raises ValueError if the maxsize parameter is incorrect.

add_logging(*writer*, *write_period=10*)
Adds ability to monitor queue sizes and fetch times.

Will try to import tensorflow and throw a warnings.warn if it couldn't.

Parameters

- **file_writer** (*tensorflow FileWriter object*) – Uses this object to write out summaries.

- **write_period** (*int*) – After how many calls to get_batch should we write to the logger.

epoch_count
Returns what epoch we are currently at

epoch_size
The epoch size (as interpreted from the File Queue)

filling
Returns true if the file queue is being filled

get(*block=True*, *timeout=None*)
Get a single item from the Image Queue

get_batch(*batch_size*, *timeout=3*)
Tries to get a batch from the Queue.

If there is less than a batch of images, it will grab them all. If the epoch size was set and the tracking counter sees there are fewer than <batch_size> images until we hit an epoch, then it will cap the amount of images grabbed to reach the epoch.

Parameters

- **batch_size** (*int*) – How many samples we want to get.
- **timeout** (*bool*) – How long to wait on timeout

Returns

- **data** (*list of ndarray*) – List of numpy arrays representing the transformed images.
- **labels** (*list of ndarray or None*) – List of labels. Will be None if there were no labels in the FileQueue.

Notes

When we pull the last batch from the image queue, the property last_batch is set to true. This allows the calling function to synchronize tests with the end of an epoch.

Raises

- FileQueueNotStarted - when trying to get a batch but the file queue
- manager hasn't started.
- FileQueueDepleted - when we have hit the epoch limit.
- ImgQueueNotStarted - when trying to get a batch but no image loaders
- have started.
- queue.Empty - If timed out on trying to read an image

img_shape
Return what the image size is of the images in the queue

This may be useful to check the output shape after any preprocessing has been done.

Returns **img_size** – Returns the shape of the images in the queue or None if it could not determine what they were.

Return type list of ints or None

join()
Method to signal any threads that are filling this queue to stop.

Threads will clean themselves up if the epoch limit is reached, but in case you want to kill them manually before that, you can signal them to stop here. Note that if these threads are blocked waiting on input, they will still stay alive (and blocked) until whatever is blocking them frees up. This shouldn't be a problem though, as they will not be taking up any processing power.

If there is a file queue associated with this image queue, those threads will be stopped too.

Note: Overloads the queue join method which normally blocks until the queue has been emptied. This will return even if the queue has data in it.

killed
Returns True if the queue has been asked to die.

label_shape
Return what the label shape is of the labels in the queue

This may be useful to check the output shape after any preprocessing has been done.

Returns **label_shape** – Returns the shape of the images in the queue or None if it could not determine what they were.

Return type list of ints or None

last_batch
Check whether the previously read batch was the last batch in the epoch.

Reading this value will set it to False. This allows you to do something like this:

```
while True:
    while not train_queue.last_batch:
        data, labels = train_queue.get_batch(batch_size)

    ...
```

read_count
Returns how many images have been read from this queue.

start_loaders(*file_queue*, *num_threads=3*, *img_dir=None*, *img_size=None*, *transform=None*)
Starts the threads to load the images into the ImageQueue

Parameters

- **file_queue** (*FileQueue object*) – An instance of the file queue
- **num_threads** (*int*) – How many parallel threads to start to load the images
- **img_dir** (*str*) – Offset to add to the strings fetched from the file queue so that a call to load the file in will succeed.
- **img_size** (*tuple of (height, width) or None*) – What size to resize all the images to. If None, no resizing will be done.
- **transform** (*function handle or None*) – Pre-filtering operation to apply to the images before adding to the Image Queue. If None, no operation will be applied. Otherwise, has to be a function handle that takes the numpy array and returns the transformed image as a numpy array.

Raises ValueError: if called after take_dataset.

take_dataset(*data*, *labels=None*, *shuffle=True*, *num_threads=1*, *transform=None*, *max_epochs=inf*)

Save the image dataset to the class for feeding back later.

If we don't need a file queue (we have all the dataset in memory), we can give it to the ImgQueue class with this method. Images will still flow through the queue (so you still need to be careful about how big to set the queue's maxsize), but now the preprocessed images will be fed into the queue, ready to retrieve quickly by the main program.

Parameters

- **data** (*ndarray of floats*) – The images. Should be in the form your main program is happy to receive them in, as no reshaping will be done. For example, if the data is of shape [10000, 32, 32, 3], then we randomly sample from the zeroth axis when we call get batch.
- **labels** (*ndarray numeric or None*) – The labels. If not None, the zeroth axis has to match the size of the data array. If None, then no labels will be returned when calling get batch.
- **shuffle** (*bool*) – Normally the ordering will be done in the file queue, as we are skipping this, the ordering has to be done here. Set this to true if you want to receive samples randomly from data.
- **num_threads** (*int*) – How many threads to start to fill up the image queue with the preprocessed data.
- **transform** (*None or callable*) – Transform to apply to images. Should accept a single image (although isn't fussy about what size/shape it is in), and return a single image. This will be applied to all the images independently before putting them in the Image Queue.

Notes

Even if shuffle input is set to false, that doesn't necessarily mean that all images in the image queue will be in the same order across epochs. For example, if thread A pulls the first 100 images from the list and then thread B gets the second 100. Thread A takes slightly longer to process the images than thread B, so these get inserted into the Image Queue afterwards. Trying to synchronize across both queues could be done, but it would add unnecessary complications and overhead.

Raises AssertionError if data and labels don't match up in size.

6.2 Exceptions

exception dataset_loading.**ImgQueueNotStarted**(*value*)

Exception Raised when trying to pull from an Image queue that hasn't had its feeders started.

exception dataset_loading.**FileQueueNotStarted**(*value*)

Exception Raised when trying to pull from a File queue that hasn't had its manager started.

exception dataset_loading.**FileQueueDepleted**(*value*)

Exception Raised when the file queue has been depleted. Will be raised when the epoch limit is reached.

6.3 Dataset Specific

6.3.1 MNIST

dataset_loading.mnist.**extract_images**(*f*)

Extract the images into a 4D uint8 numpy array [index, y, x, depth].

Parameters **f** (*file object*) – file that can be passed into a gzip reader.

Returns **data**

Return type A 4D uint8 numpy array [index, y, x, depth]

Raises ValueError: If the bytestream does not start with 2051.

dataset_loading.mnist.**extract_labels**(*f*, *one_hot=False*, *num_classes=10*)

Extract the labels into a 1D uint8 numpy array [index].

Parameters

- **f** (*file object*) – A file object that can be passed into a gzip reader.
- **one_hot** (*bool*) – Does one hot encoding for the result.
- **num_classes** (*int*) – Number of classes for the one hot encoding.

Returns **labels**

Return type a 1D uint8 numpy array.

Raises ValueError: If the bystream doesn't start with 2049.

dataset_loading.mnist.**get_mnist_queues**(*data_dir*, *val_size=2000*, *transform=None*, *maxsize=10000*, *num_threads=(2, 2, 2)*, *max_epochs=inf*, *get_queues=(True, True, True)*, *one_hot=True*, *download=False*, *_rand_data=False*)

Get Image queues for MNIST

MNIST is a small dataset. This function loads it into memory and creates several ImgQueue to feed the training, testing and validation data through to the main function. Preprocessing can be done by providing a callable to the transform parameter. Note that by default, the black and white MNIST images will be returned as a [28, 28, 1] shape numpy array. You can of course modify this with the transform function.

Parameters

- **data_dir** (*str*) – Path to the folder containing the cifar data. For cifar10, this should be the path to the folder called 'cifar-10-batches-py'. For cifar100, this should be the path to the folder 'cifar-100-python'.
- **val_size** (*int*) – How big you want the validation set to be. Will be taken from the end of the train data.
- **transform** (*None or callable or tuple of callables*) – Callable function that accepts a numpy array representing **one** image, and transforms it/preprocesses it. E.g. you may want to remove the mean and divide by standard deviation before putting into the queue. If tuple of callables, needs to be of length 3 and should be in the order (train_transform, test_transform, val_transform). Setting it to None means no processing will be done before putting into the image queue.
- **maxsize** (*int or tuple of 3 ints*) – How big the image queues will be. Increase this if your main program is chewing through the data quickly, but increasing it will also

mean more memory is taken up. If tuple of ints, needs to be length 3 and of the form (train_qsize, test_qsize, val_qsize).

- **num_threads** (*int or tuple of 3 ints*) – How many threads to use for the train, test and validation threads (if tuple, needs to be of length 3 and in that order).
- **max_epochs** (*int*) – How many epochs to run before returning FileQueueDepleted exceptions
- **get_queues** (*tuple of 3 bools*) – In case you only want to have training data, or training and validation, or any subset of the three queues, you can mask the individual queues by putting a False in its position in this tuple of 3 bools.
- **one_hot** (*bool*) – True if you want the labels pushed into the queue to be a one-hot vector. If false, will push in a one-of-k representation.
- **download** (*bool*) – True if you want the dataset to be downloaded for you. It will be downloaded into the data_dir provided in this case.

Returns

- **train_queue** (ImgQueue instance or None) – Queue with the training data in it. None if get_queues[0] == False
- **test_queue** (ImgQueue instance or None) – Queue with the test data in it. None if get_queues[1] == False
- **val_queue** (ImgQueue instance or None) – Queue with the validation data in it. Will be None if the val_size parameter was 0 or get_queues[2] == False

Notes

If the max_epochs paramter is set to a finite amount, then when the queues run out of data, they will raise a dataset_loading.FileQueueDepleted exception.

dataset_loading.mnist.**load_mnist_data**(*data_dir*, *val_size=2000*, *one_hot=True*, *download=False*)

Load mnist data

Parameters

- **data_dir** (*str*) – Path to the folder with the mnist files in them. These should be the gzip files downloaded from yann.lecun.com
- **val_size** (*int*) – Size of the validation set.
- **one_hot** (*bool*) – True to return one hot labels
- **download** (*bool*) – True if you don't have the data and want it to be downloaded for you.

Returns

- **trainx** (*ndarray*) – Array containing training images. There will be 60000 - *val_size* images in this.
- **trainy** (*ndarray*) – Array containing training labels. These will be one hot if the one_hot parameter was true, otherwise the standard one of k.
- **testx** (*ndarray*) – Array containing test images. There will be 10000 test images in this.
- **testy** (*ndarray*) – Test labels
- **valx** (*ndarray*) – Array containing validation images. Will be None if val_size was 0.

- **valy** (*ndarray*) – Array containing validation labels. Will be None if val_size was 0.

6.3.2 CIFAR

dataset_loading.cifar.**load_cifar_data**(*data_dir*, *cifar10=True*, *val_size=2000*, *one_hot=True*, *download=False*)

Load cifar10 or cifar100 data

Parameters

- **data_dir** (*str*) – Path to the folder with the cifar files in them. These should be the python files as downloaded from cs.toronto
- **cifar10** (*bool*) – True if cifar10, false if cifar100
- **val_size** (*int*) – Size of the validation set.
- **one_hot** (*bool*) – True to return one hot labels
- **download** (*bool*) – True if you don't have the data and want it to be downloaded for you.

Returns

- **trainx** (*ndarray*) – Array containing training images. There will be 50000 - *val_size* images in this.
- **trainy** (*ndarray*) – Array containing training labels. These will be one hot if the one_hot parameter was true, otherwise the standard one of k.
- **testx** (*ndarray*) – Array containing test images. There will be 10000 test images in this.
- **testy** (*ndarray*) – Test labels
- **valx** (*ndarray*) – Array containing validation images. Will be None if val_size was 0.
- **valy** (*ndarray*) – Array containing validation labels. Will be None if val_size was 0.

dataset_loading.cifar.**get_cifar_queues**(*data_dir*, *cifar10=True*, *val_size=2000*, *transform=None*, *maxsize=10000*, *num_threads=(2, 2, 2)*, *max_epochs=inf*, *get_queues=(True, True, True)*, *one_hot=True*, *download=False*, *_rand_data=False*)

Get Image queues for CIFAR

CIFAR10/100 are both small datasets. This function loads them both into memory and creates several `ImgQueue` instances to feed the training, testing and validation data through to the main function. Preprocessing can be done by providing a callable to the transform parameter. Note that by default, the CIFAR images returned will be of shape [32, 32, 3] but this of course can be changed by the transform function.

Parameters

- **data_dir** (*str*) – Path to the folder containing the cifar data. For cifar10, this should be the path to the folder called 'cifar-10-batches-py'. For cifar100, this should be the path to the folder 'cifar-100-python'.
- **cifar10** (*bool*) – True if we are using cifar10.
- **val_size** (*int*) – How big you want the validation set to be. Will be taken from the end of the train data.
- **transform** (*None or callable or tuple of callables*) – Callable function that accepts a numpy array representing **one** image, and transforms it/preprocesses it. E.g. you may want to remove the mean and divide by standard deviation before putting

into the queue. If tuple of callables, needs to be of length 3 and should be in the order (train_transform, test_transform, val_transform). Setting it to None means no processing will be done before putting into the image queue.

- **maxsize** (*int or tuple of 3 ints*) – How big the image queues will be. Increase this if your main program is chewing through the data quickly, but increasing it will also mean more memory is taken up. If tuple of ints, needs to be length 3 and of the form (train_qsize, test_qsize, val_qsize).
- **num_threads** (*int or tuple of 3 ints*) – How many threads to use for the train, test and validation threads (if tuple, needs to be of length 3 and in that order).
- **max_epochs** (*int*) – How many epochs to run before returning FileQueueDepleted exceptions
- **get_queues** (*tuple of 3 bools*) – In case you only want to have training data, or training and validation, or any subset of the three queues, you can mask the individual queues by putting a False in its position in this tuple of 3 bools.
- **one_hot** (*bool*) – True if you want the labels pushed into the queue to be a one-hot vector. If false, will push in a one-of-k representation.
- **download** (*bool*) – True if you want the dataset to be downloaded for you. It will be downloaded into the data_dir provided in this case.

Returns

- **train_queue** (ImgQueue instance or None) – Queue with the training data in it. None if get_queues[0] == False
- **test_queue** (ImgQueue instance or None) – Queue with the test data in it. None if get_queues[1] == False
- **val_queue** (ImgQueue instance or None) – Queue with the validation data in it. Will be None if the val_size parameter was 0 or get_queues[2] == False

Notes

If the max_epochs paramter is set to a finite amount, then when the queues run out of data, they will raise a dataset_loading.FileQueueDepleted exception.

6.3.3 PASCAL

dataset_loading.pascal.**img_sets**()
List all the image sets from Pascal VOC. Don't bother computing this on the fly, just remember it. It's faster.

dataset_loading.pascal.**load_pascal_data**(*data_dir*, *max_epochs=None*, *thread_count=3*, *im-size=(128*, *128)*)
Will use a filename queue and img_queue and load the data

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index