deepdish Documentation

Release 0.3.5.git

Gustav Larsson, Mark Stoehr

October 01, 2016

Contents

Contents:

CHAPTER 1

Tutorials

1.1 Saving and loading data

Saving and loading Python data types is one of the most common operations when putting together experiments. One method is to use pickling, but this is not compatible between Python 2 and 3, and the files cannot be easily inspected or shared with other programming languages.

Deepdish has a function that converts your Python data type into a native HDF5 hierarchy. It stores dictionaries, SimpleNamespaces (for versions of Python that support them), values, strings and numpy arrays very naturally. It can also store lists and tuples, but it's not as natural, so prefer numpy arrays whenever possible. Here's an example saving and HDF5 using *deepdish.io.save()*:

```
>>> import deepdish as dd
>>> d = {'foo': np.arange(10), 'bar': np.ones((5, 4, 3))}
>>> dd.io.save('test.h5', d)
```

It will try its best to save it in a way native to HDF5:

```
$ h5ls test.h5
bar                      Dataset {5, 4, 3}
foo                      Dataset {10}
```

We also offer our own version of `h5ls` that works really well with deepdish saved HDF5 files:

```
$ ddls test.h5
/bar                       array (5, 4, 3) [float64]
/foo                       array (10,) [int64]
```

We can now reconstruct the dictionary from the file using *deepdish.io.load()*:

```
>>> d = dd.io.load('test.h5')
```

1.1.1 Dictionaries

Dictionaries are saved as HDF5 groups:

```
>>> d = {'foo': {'bar': np.arange(10), 'baz': np.zeros(3)}, 'qux': np.ones(12)}
>>> dd.io.save('test.h5', d)
```

Resulting in:

```
$ h5ls -r test.h5
/                        Group
/foo                     Group
/foo/bar                 Dataset {10}
/foo/baz                 Dataset {3}
/qux                     Dataset {12}
```

Again, we can use the deepdish tool for better inspection:

```
$ ddls test.h5
/foo                       dict
/foo/bar                   array (10,) [int64]
/foo/baz                   array (3,) [float64]
/qux                       array (12,) [float64]
```

1.1.2 SimpleNamespaces

SimpleNamespaces work almost identically to dictionaries and are available in Python 3.3 and later. Note that for versions of Python that do not support SimpleNamespaces, deepdish will load them in as dictionaries.

Like dictionaries, SimpleNamespaces are saved as HDF5 groups:

```
>>> from types import SimpleNamespace as NS
>>> d = NS(foo=NS(bar=np.arange(10), baz=np.zeros(3)), qux=np.ones(12))
>>> dd.io.save('test.h5', d)
```

For h5ls, the results are identical to the Dictionary example:

```
$ h5ls -r test.h5
/                        Group
/foo                     Group
/foo/bar                 Dataset {10}
/foo/baz                 Dataset {3}
/qux                     Dataset {12}
```

Again, we can use the deepdish tool for better inspection. For a version of Python that supports SimpleNamespaces:

```
$ ddls test.h5
/                          SimpleNamespace
/foo                       SimpleNamespace
/foo/bar                   array (10,) [int64]
/foo/baz                   array (3,) [float64]
/qux                       array (12,) [float64]
```

For a version of Python that doesn't support SimpleNamespaces, dictionaries are used:

```
$ ddls test.h5
/foo                       dict
/foo/bar                   array (10,) [int64]
/foo/baz                   array (3,) [float64]
/qux                       array (12,) [float64]
```

1.1.3 Numpy arrays

Numpy arrays of any numeric data type are natively stored in HDF5:

```
>>> d = {'a': np.arange(5),
...      'b': np.array([1.2, 2.3, 3.4]),
...      'c': np.ones(3, dtype=np.int8)}
```

We can inspect the actual values:

```
$ h5ls -d test.h5
a                        Dataset {5}
    Data:
        (0) 0, 1, 2, 3, 4
b                        Dataset {3}
    Data:
        (0) 1.2, 2.3, 3.4
c                        Dataset {3}
    Data:
        (0) 1, 1, 1
```

1.1.4 Basic data types

Basic Python data types are stored as attributes or empty groups:

```
>>> d = {'a': 10, 'b': 'test', 'c': None}
>>> dd.io.save('test.h5', d)
```

We might not see them through `h5ls`:

```
$ h5ls test.h5
c                        Group
```

This is where `ddls` excels:

```
$ ddls test.h5
/a                         10 [int64]
/b                         'test' (4) [unicode]
/c                         None [python]
```

Since *c* is specific to Python, it is stored as an empty group with meta information. The values *a* and *b* however are stored natively as HDF5 attributes:

```
$ ptdump -a test.h5
/ (RootGroup) ''
  /._v_attrs (AttributeSet), 7 attributes:
   [CLASS := 'GROUP',
    DEEPDISH_IO_VERSION := 10,
    PYTABLES_FORMAT_VERSION := '2.1',
    TITLE := '',
    VERSION := '1.0',
    a := 10,
    b := 'test']
/c (Group) 'nonetype:'
  /c._v_attrs (AttributeSet), 3 attributes:
   [CLASS := 'GROUP',
    TITLE := 'nonetype:',
    VERSION := '1.0']
```

Note that these are still somewhat awkwardly stored, so always prefer using numpy arrays to store numeric values.

1.1.5 Lists and tuples

Lists and tuples are shoehorned into the HDF5 key-value structure by letting each element be its own group (or attribute, depending on the type of the element):

```
>>> x = [{'foo': 10}, {'bar': 20}, 30]
>>> dd.io.save('test.h5', x)
```

The first two elements are stored as `'i0'` and `'i1'`. The third is stored as an attribute and thus not directly visible by `h5ls`. However, `ddls` will show it:

```
$ ddls test.h5
/data*                      list
/data/i0                    dict
/data/i0/foo                10 [int64]
/data/i1                    dict
/data/i1/bar                20 [int64]
/data/i2                    30 [int64]
```

Note that this is awkward and if the list is long you easily hit HDF5's limitation on the number of groups. Therefore, if your list is numeric, always make it a numpy array first! The asterisk on the "/data" group indicates that the top level variable that was saved was not a dict or a SimpleNamespace; during load, deepdish will unpack "/data" so that the saved variable is returned. See *Fake top-level group*.

1.1.6 Pandas data structures

The pandas data structures `DataFrame`, `Series` and `Panel` are natively supported. This is thanks to pandas already providing support for this with the same PyTables backend as deepdish:

```
import pandas as pd
df = pd.DataFrame({'int': np.arange(3), 'name': ['zero', 'one', 'two']})

dd.io.save('test.h5', df)
```

We can inspect this as usual:

```
$ ddls test.h5
/data*                      DataFrame (2, 3)
```

If you are curious of how pandas stores this, we can tell `ddls` to forget it knows how to read data frames by invoking the `--raw` command:

```
$ ddls test.h5 --raw
/data*                      dict
/data/axis0                 array (2,) [|S4]
/data/axis0_variety         'regular' (7) [unicode]
/data/axis1                 array (3,) [int64]
/data/axis1_variety         'regular' (7) [unicode]
/data/block0_items          array (1,) [|S3]
/data/block0_items_vari...  'regular' (7) [unicode]
/data/block0_values         array (3, 1) [int64]
/data/block1_items          array (1,) [|S4]
/data/block1_items_vari...  'regular' (7) [unicode]
/data/block1_values         pickled [object]
/data/encoding              'UTF-8' (5) [unicode]
/data/nblocks               2 [int64]
/data/ndim                  2 [int64]
```

```
/data/pandas_type           'frame' (5) [unicode]
/data/pandas_version        '0.15.2' (6) [unicode]
```

1.1.7 Sparse matrices

Scipy offers several types of sparse matrices, of which deepdish can save the types BSR, COO, CSC, CSR and DIA. The types DOK and LIL are currently not supported (note that these two types are mainly for incrementally building sparse matrices anyway).

Just like with pandas data types, you can inspect the storage format using `ddls --raw`.

1.1.8 Compression

The way sparse matrices are stored in deepdish are identical to how they are represented in Numpy, meaning there is no conversion time and the storage is compact. The further minimize the disk space, deepdish offer several means of compressing the data (all thanks to the powerful PyTables backend).

Here is a comparison on a large (100 billion elements) and sparse (0.01% sparsity) CSR matrix:

Method	Compression	Space (MB)	Write time (s)	Read time (s)
scipy's mmwrite	N	145	79	40
numpy's save	N	134	1.36	0.75
pickle	N	115	0.63	0.17
deepdish (no compression)	N	115	0.52	0.17
numpy's savez_compressed	Y	32	8.88	1.33
pickle (gzip)	Y	29	5.19	0.86
deepdish (blosc)	Y	24	0.36	0.37
deepdish (zlib)	Y	21	9.01	0.83

This particular matrix had only nonzero elements that were set to 1, which meant even more compression could be applied. The default compression in deepdish is zlib, since it is widely supported and means your HDF5 files saved with deepdish can be universally read. However, blosc is clearly a much better choice, so if interoperability (e.g. with MATLAB) is not a priority, we encourage you to change the default. You can do this by placing the following in the file `~/.deepdish.conf`:

```
[io]
compression: blosc
```

To change the default to no compression, use `compression: none`.

1.1.9 Quick inspection

Using `ddls` gives you a quick overview of your data. You can also use it to print specific entries from the command line:

```
$ ddls test.h5 -i /foo
[0 1 2 3 4 5 6 7 8 9]
```

Adding `--ipython` will start an IPython session that comes pre-loaded with the selected variable loaded into the variable `data`. This can be used even without `-i`, in which case the whole file is loaded into `data`.

1.1.10 Fake top-level group

Even if the entry object is not a dictionary or SimpleNamespace, HDF5 forces us to create a top-level group to put it in. This group will be called `data` and marked using hidden attributes as fake so that a dictionary or SimpleNamespace is not added when loaded:

```
dd.io.save('test.h5', [np.arange(5), 100])
```

Note that `ddls` will let you know this group is fake by adding a star after `/data`:

```
$ ddls test.h5
/data*                      list
/data/i0                    array (5,) [int64]
/data/i1                    100 [int64]
```

1.1.11 Partial loading

A specific level can be loaded as follows:

```
dd.io.save('test.h5', dict(foo=dict(bar=np.ones((10, 5)))))

bar = dd.io.load('test.h5', '/foo/bar')
```

You can even load slices of arrays:

```
bar_slice = dd.io.load('test.h5', '/foo/bar', sel=dd.aslice[:5, -2:])
```

The file will never be read in full, not even the array, so this technique can be used to step through very large arrays.

To load multiple groups at once, use a list of strings:

```
data, label = dd.io.load('training.h5', ['/data', '/label'])
```

1.1.12 Pickled objects

Some objects cannot be saved natively as HDF5, such as object classes. Our suggestion is to convert classes to to dictionary-like structures first, but sometimes it can be nice to be able to dump anything into a file. This is why deepdish also offers pickling as a last resort:

```
import deepdish as dd

class Foo(object):
    pass

foo = Foo()
dd.io.save('test.h5', dict(foo=foo))
```

Inspecting this file will yield:

```
$ ddls test.h5
/foo                        pickled [object]
```

Note that the class *Foo* has to be defined in the file that calls `dd.io.load`.

Avoid relying on pickling, since it hurts the interoperability provided by deepdish's HDF5 saving. Each pickled object will raise a `DeprecationWarning`, so call Python with `-Wall` to make sure you aren't implicitly pickling something. You can of course also use `ddls` to inspect the file to make sure nothing is pickled.

If deepdish fatally fails to save an object, you should first report this as an issue on GitHub. As a quick fix, you can force pickling by wrapping the object in `deepdish.io.ForcePickle()`:

```
dd.io.save('test.h5', {'foo': dd.io.ForcePickle('pickled string')})
```

1.1.13 Class instances

Storing classes can be done by converting them to and from dictionary structures. This is a bit more work than straight up pickling, but the benefit is that they are inspectable from outside Python, and compatible between Python 2 and 3 (which pickled classes are not!). This process can be facilitated by subclassing *deepdish.util.SaveableRegistry*:

```
import deepdish as dd

class Foo(dd.util.SaveableRegistry):
    def __init__(self, x):
        self.x = x

    @classmethod
    def load_from_dict(self, d):
        obj = Foo(d['x'])
        return obj

    def save_to_dict(self):
        return {'x': self.x}


@Foo.register('bar')
class Bar(Foo):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    @classmethod
    def load_from_dict(self, d):
        obj = Bar(d['x'], d['y'])
        return obj

    def save_to_dict(self):
        return {'x': self.x, 'y': self.y}
```

Now, we can save an instane of *Foo* directly to an HDF5 file by:

```
>>> f = Foo(10)
>>> f.save('foo.h5')
```

And restore it by:

```
>>> f = Foo.load('foo.h5')
```

In the example, we also showed how we can subclass *Foo* as *Bar*. Now, we can do:

```
>>> b = Bar(10, 20)
>>> b.save('bar.h5')
```

Along with *x* and *y*, it will save `'bar'` as meta-information (since we called `Foo.register('bar')`). What this means is that we can reconstruct the instance from:

```
>>> b = Foo.load('bar.h5')
```

Note that we did not need to call *Bar.load* (although it would work too), which makes it very easy to load subclassed instances of various kinds. The registered named can be accessed through:

```
>>> b.name
'bar'
```

To give the base class a name, we can add `dd.util.SaveableRegistry.register('foo')` before the class definition.

1.1.14 Soft Links

In Python, many names can be bound to the same object. Deepdish accounts for this for many objects (dictionaries, lists, numpy arrays, pandas dataframes, SimpleNamespaces, etc) by using HDF5 soft links. This means the object itself is only written once and that these relationships are preserved upon loading. Recursion inside objects is also handled via soft links.

Here is an example where soft links are used for both purposes:

```
>>> import deepdish as dd
>>> ones = np.ones((5, 4, 3))
>>> d = {'foo': np.arange(10), 'bar': ones, 'baz': ones}
>>> d['self'] = d   # to demonstrate recursion
>>> dd.io.save('test.h5', d)
```

Soft links are native to HDF5:

```
$ h5ls test.h5
bar                      Dataset {5, 4, 3}
baz                      Soft Link {/bar}
foo                      Dataset {10}
self                     Soft Link {/}
```

Notice that the *ones* 3D array was only written once and that `self` is a link to the top level group. With `ddls`:

```
$ ddls test.h5
/bar                        array (5, 4, 3) [float64]
/baz                        link -> /bar [SoftLink]
/foo                        array (10,) [int64]
/self                       link -> / [SoftLink]
```

Verify that `d['bar']` and `d['baz']` refer to the same object:

```
>>> d = dd.io.load('test.h5')
>>> d['bar'] is d['baz']
True
```

Also verify that `d['self']` is `d`:

```
>>> d['self'] is d
True
```

1.2 Parallelization

This module provides a method of doing parallelization using MPI. It uses *mpi4py*, but provides convenience functions that make parallelization much easier.

If you have the following file (*double.py*):

```
def compute(x):
    return 2 * x

if __name__ == '__main__':
    values = range(20)

    for x in map(compute, values):
        print(x)
```

You can parallelize the computation by replacing it with the following (*double_mpi.py*):

```
import deepdish as dd

def compute(x):
    return 2 * x

if dd.parallel.main(__name__):
    values = range(20)

    for x in dd.parallel.imap(compute, values):
        print(x)
```

And run it with:

```
$ mpirun -n 8 python double_mpi.py
```

The way it works is that *deepdish.parallel.main()* will be a gate-keeper that only lets through rank 0. The rest of the nodes will stand idly by. Then, when *deepdish.parallel.imap()* is called, the computation is sent to the worker nodes, computed, and finally sent back to rank 0. Note that the rank 0 node is not given a task since it needs to be ready to execute the body of the for loop (the *imap* function will yield values as soon as ready). If the body of the for loop is light on computation, you might want to tell *mpirun* that you want one more job than your cores.

The file *double_mpi.py* can also be run without *mpirun* as well.

Note that if your high performance cluster has good MPI support, this will allow you to parallelize not only across cores, but across machines.

1.2.1 Multiple arguments

If you have multiple arguments, you can use *deepdish.parallel.starmap()* to automatically unpack them. Note that *starmap* also returns a generator that will yield results as soon as they are done, so to gather all into a list we have to run it through *list* before giving it to *concatenate*:

```
import deepdish as dd
import numpy as np

def compute(batch, x):
    print('Processing batch', batch, 'on node', dd.parallel.rank())
    return 2 * x

if dd.parallel.main(__name__):
    x = np.arange(100)
    batches = np.array_split(x, 10)
    args = ((batch, x) for batch, x in enumerate(batches))

    # Execute and combine results
```

```
    y = np.concatenate(list(dd.parallel.starmap(compute, args)))
    print(y)
```

We are also showing that you can print the rank using *deepdish.parallel.rank()*.

1.2.2 Unordered

Sometimes we don't care what order the jobs get processed in, in which case we can use *deepdish.parallel.imap_unordered()* or *deepdish.parallel.starmap_unordered()*. This is great if we are doing a commutative reduction on the results or if we are processing or testing independent samples. The results will be yielded as soon as any batch is done, which means more responsive output and a smaller memory footprint than its ordered counterpart. However, this means that you won't necessarily know which batch completed, so you might have to put the batch number in as one of the arguments and return it. In this example, we do something similar which is to run the indices through the *compute* function:

```
import deepdish as dd
import numpy as np

def compute(indices, x):
    return indices, 2 * x

if dd.parallel.main(__name__):
    x = np.arange(100) * 10
    index_batches = np.array_split(np.arange(len(x)), 10)
    args = ((indices, x[indices]) for indices in index_batches)

    y = np.zeros_like(x)
    for indices, batch_y in dd.parallel.starmap_unordered(compute, args):
        print('Finished indices', indices)
        y[indices] = batch_y

    print(y)
```

For more information, see the *deepdish.parallel* API documentation.

CHAPTER 2

API Reference

2.1 Core functions

deepdish.**bytesize**(*arr*)
Returns the memory byte size of a Numpy array as an integer.

deepdish.**memsize**(*arr*)
Returns the required memory of a Numpy array as a humanly readable string.

deepdish.**span**(*arr*)
Calculate and return the mininum and maximum of an array.

Parameters **arr** (*ndarray*) – Numpy array.

Returns

- **min** (*dtype*) – Minimum of array.
- **max** (*dtype*) – Maximum of array.

deepdish.**apply_once**(*func*, *arr*, *axes*, *keepdims=True*)
Similar to *numpy.apply_over_axes*, except this performs the operation over a flattened version of all the axes, meaning that the function will only be called once. This only makes a difference for non-linear functions.

Parameters

- **func** (*callback*) – Function that operates well on Numpy arrays and returns a single value of compatible dtype.
- **arr** (*ndarray*) – Array to do operation over.
- **axes** (*int or iterable*) – Specifies the axes to perform the operation. Only one call will be made to *func*, with all values flattened.
- **keepdims** (*bool*) – By default, this is True, so the collapsed dimensions remain with length 1. This is simlar to *numpy.apply_over_axes* in that regard. If this is set to False, the dimensions are removed, just like when using for instance *numpy.sum* over a single axis. Note that this is safer than subsequently calling squeeze, since this option will preserve length-1 dimensions that were not operated on.

Examples

```
>>> import deepdish as dd
>>> import numpy as np
>>> rs = np.random.RandomState(0)
>>> x = rs.uniform(size=(10, 3, 3))
```

Image that you have ten 3x3 images and you want to calculate each image's intensity standard deviation:

```
>>> np.apply_over_axes(np.std, x, [1, 2]).ravel()
array([ 0.06056838,  0.08230712,  0.08135083,  0.09938963,  0.08533604,
        0.07830725,  0.066148  ,  0.07983019,  0.08134123,  0.01839635])
```

This is the same as `x.std(1).std(1)`, which is not the standard deviation of all 9 pixels together. To fix this we can flatten the pixels and try again:

```
>>> x.reshape(10, 9).std(axis=1)
array([ 0.17648981,  0.32849108,  0.29409526,  0.25547501,  0.23649064,
        0.26928468,  0.20081239,  0.33052397,  0.29950855,  0.26535717])
```

This is exactly what this function does for you:

```
>>> dd.apply_once(np.std, x, [1, 2], keepdims=False)
array([ 0.17648981,  0.32849108,  0.29409526,  0.25547501,  0.23649064,
        0.26928468,  0.20081239,  0.33052397,  0.29950855,  0.26535717])
```

deepdish.**tupled_argmax**(*a*)

Argmax that returns an index tuple. Note that *numpy.argmax* will return a scalar index as if you had flattened the array.

Parameters **a** (*array_like*) – Input array.

Returns **index** – Tuple of index, even if *a* is one-dimensional. Note that this can immediately be used to index *a* as in `a[index]`.

Return type tuple

Examples

```
>>> import numpy as np
>>> import deepdish as dd
>>> a = np.arange(6).reshape(2,3)
>>> a
array([[0, 1, 2],
       [3, 4, 5]])
>>> dd.tupled_argmax(a)
(1, 2)
```

deepdish.**timed**(**args*, ***kwds*)

Context manager to make it easy to time the execution of a piece of code. This timer will never run your code several times and is meant more for simple in-production timing, instead of benchmarking. Reports the wall-clock time (using *time.time*) and not the processor time.

Parameters

- **name** (*str*) – Name of the timing block, to identify it.
- **file** (*file handler*) – Which file handler to print the results to. Default is standard output. If a numpy array and size 1 is given, the time in seconds will be stored inside it. Ignored if *callback* is set.

- **callback** (*callable*) – This offer even more flexibility than *file*. The callable will be called at the end of the execution with a single floating point argument with the elapsed time in seconds.

Examples

```
>>> import deepdish as dd
>>> import time
```

The *timed* function is a context manager, so everything inside the `with` block will be timed. The results will be printed by default to standard output:

```
>>> with dd.timed('Sleep'):
...     time.sleep(1)
[timed] Sleep: 1.001035451889038 s
```

Using the *callback* parameter, we can accumulate multiple runs into a list:

```
>>> times = []
>>> for i in range(3):
...     with dd.timed(callback=times.append):
...         time.sleep(1)
>>> times
[1.0035350322723389, 1.0035550594329834, 1.0039470195770264]
```

2.2 IO (deepdish.io)

See the *Saving and loading data* chapter for a tutorial.

deepdish.io.**save**(*path*, *data*, *compression='default'*)

Save any Python structure to an HDF5 file. It is particularly suited for Numpy arrays. This function works similar to `numpy.save`, except if you save a Python object at the top level, you do not need to issue `data.flat[0]` to retrieve it from inside a Numpy array of type `object`.

Some types of objects get saved natively in HDF5. The rest get serialized automatically. For most needs, you should be able to stick to the natively supported types, which are:

- Dictionaries
- Short lists and tuples (<256 in length)
- Basic data types (including strings and None)
- Numpy arrays
- Pandas `DataFrame`, `Series`, and `Panel`
- SimpleNamespaces (for Python >= 3.3, but see note below)

A recommendation is to always convert your data to using only these types That way your data will be portable and can be opened through any HDF5 reader. A class that helps you with this is *deepdish.util.Saveable*.

Lists and tuples are supported and can contain heterogeneous types. This is mostly useful and plays well with HDF5 for short lists and tuples. If you have a long list (>256) it will be serialized automatically. However, in such cases it is common for the elements to have the same type, in which case we strongly recommend converting to a Numpy array first.

Note that the SimpleNamespace type will be read in as dictionaries for earlier versions of Python.

This function requires the PyTables module to be installed.

You can change the default compression method to `blosc` (much faster, but less portable) by creating a `~/.deepdish.conf` with:

```
[io]
    compression: blosc
```

This is the recommended compression method if you plan to use your HDF5 files exclusively through deepdish (or PyTables).

Parameters

- **path** (*string*) – Filename to which the data is saved.
- **data** (*anything*) – Data to be saved. This can be anything from a Numpy array, a string, an object, or a dictionary containing all of them including more dictionaries.
- **compression** (*string or tuple*) – Set compression method, choosing from *blosc*, *zlib*, *lzo*, *bzip2* and more (see PyTables documentation). It can also be specified as a tuple (e.g. `('blosc', 5)`), with the latter value specifying the level of compression, choosing from 0 (no compression) to 9 (maximum compression). Set to *None* to turn off compression. The default is *zlib*, since it is highly portable; for much greater speed, try for instance *blosc*.

See also:

load()

deepdish.io.**load**(*path*, *group=None*, *sel=None*, *unpack=False*)

Loads an HDF5 saved with *save*.

This function requires the PyTables module to be installed.

Parameters

- **path** (*string*) – Filename from which to load the data.
- **group** (*string or list*) – Load a specific group in the HDF5 hierarchy. If *group* is a list of strings, then a tuple will be returned with all the groups that were specified.
- **sel** (*slice or tuple of slices*) – If you specify *group* and the target is a numpy array, then you can use this to slice it. This is useful for opening subsets of large HDF5 files. To compose the selection, you can use *deepdish.aslice*.
- **unpack** (*bool*) – If True, a single-entry dictionaries will be unpacked and the value will be returned directly. That is, if you save `dict(a=100)`, only `100` will be loaded.

Returns **data** – Hopefully an identical reconstruction of the data that was saved.

Return type anything

See also:

save()

2.3 Utilities (`deepdish.util`)

deepdish.util.**pad**(*data*, *padwidth*, *value=0.0*)

Pad an array with a specific value.

Parameters

- **data** (*ndarray*) – Numpy array of any dimension and type.

- **padwidth** (*int or tuple*) – If int, it will pad using this amount at the beginning and end of all dimensions. If it is a tuple (of same length as *ndim*), then the padding amount will be specified per axis.
- **value** (*data.dtype*) – The value with which to pad. Default is `0.0`.

See also:

pad_to_size(), *pad_repeat_border()*, *pad_repeat_border_corner()*

Examples

```
>>> import deepdish as dd
>>> import numpy as np
```

Pad an array with zeros.

```
>>> x = np.ones((3, 3))
>>> dd.util.pad(x, (1, 2), value=0.0)
array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  1.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  1.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  1.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.]])
```

deepdish.util.**pad_to_size**(*data*, *shape*, *value=0.0*)

This is similar to *pad*, except you specify the final shape of the array.

Parameters

- **data** (*ndarray*) – Numpy array of any dimension and type.
- **shape** (*tuple*) – Final shape of padded array. Should be tuple of length `data.ndim`. If it has to pad unevenly, it will pad one more at the end of the axis than at the beginning. If a dimension is specified as `-1`, then it will remain its current size along that dimension.
- **value** (*data.dtype*) – The value with which to pad. Default is `0.0`. This can even be an array, as long as `pdata[:] = value` is valid, where `pdata` is the size of the padded array.

Examples

```
>>> import deepdish as dd
>>> import numpy as np
```

Pad an array with zeros.

```
>>> x = np.ones((4, 2))
>>> dd.util.pad_to_size(x, (5, 5))
array([[ 0.,  1.,  1.,  0.,  0.],
       [ 0.,  1.,  1.,  0.,  0.],
       [ 0.,  1.,  1.,  0.,  0.],
       [ 0.,  1.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
```

deepdish.util.**pad_repeat_border**(*data*, *padwidth*)

Similar to *pad*, except the border value from `data` is used to pad.

Parameters

- **data** (*ndarray*) – Numpy array of any dimension and type.
- **padwidth** (*int or tuple*) – If int, it will pad using this amount at the beginning and end of all dimensions. If it is a tuple (of same length as *ndim*), then the padding amount will be specified per axis.

Examples

```
>>> import deepdish as dd
>>> import numpy as np
```

Pad an array by repeating its borders:

```
>>> shape = (3, 4)
>>> x = np.arange(np.prod(shape)).reshape(shape)
>>> dd.util.pad_repeat_border(x, 2)
array([[ 0,  0,  0,  1,  2,  3,  3,  3],
       [ 0,  0,  0,  1,  2,  3,  3,  3],
       [ 0,  0,  0,  1,  2,  3,  3,  3],
       [ 4,  4,  4,  5,  6,  7,  7,  7],
       [ 8,  8,  8,  9, 10, 11, 11, 11],
       [ 8,  8,  8,  9, 10, 11, 11, 11],
       [ 8,  8,  8,  9, 10, 11, 11, 11]])
```

deepdish.util.**pad_repeat_border_corner**(*data*, *shape*)

Similar to *pad_repeat_border*, except the padding is always done on the upper end of each axis and the target size is specified.

Parameters

- **data** (*ndarray*) – Numpy array of any dimension and type.
- **shape** (*tuple*) – Final shape of padded array. Should be tuple of length `data.ndim`. If it has to pad unevenly, it will pad one more at the end of the axis than at the beginning.

Examples

```
>>> import deepdish as dd
>>> import numpy as np
```

Pad an array by repeating its upper borders.

```
>>> shape = (3, 4)
>>> x = np.arange(np.prod(shape)).reshape(shape)
>>> dd.util.pad_repeat_border_corner(x, (5, 5))
array([[  0.,   1.,   2.,   3.,   3.],
       [  4.,   5.,   6.,   7.,   7.],
       [  8.,   9.,  10.,  11.,  11.],
       [  8.,   9.,  10.,  11.,  11.],
       [  8.,   9.,  10.,  11.,  11.]])
```

class deepdish.util.**Saveable**

Key-value coding interface for classes. Generally, this is an interface that make it possible to access instance members through keys (strings), instead of through named variables. What this interface enables, is to save and load an instance of the class to file. This is done by encoding it into a dictionary, or decoding it from a dictionary. The dictionary is then saved/loaded using *deepdish.io.save*.

classmethod load(*path*)
Loads an instance of the class from a file.

Parameters **path** (*str*) – Path to an HDF5 file.

Examples

This is an abstract data type, but let us say that `Foo` inherits from `Saveable`. To construct an object of this class from a file, we do:

```
>>> foo = Foo.load('foo.h5')
```

classmethod load_from_dict(*d*)
Overload this function in your subclass. It takes a dictionary and should return a constructed object.

When overloading, you have to decorate this function with `@classmethod`.

Parameters **d** (*dict*) – Dictionary representation of an instance of your class.

Returns **obj** – Returns an object that has been constructed based on the dictionary.

Return type object

save(*path*)
Saves an instance of the class using *deepdish.io.save*.

Parameters **path** (*str*) – Output path to HDF5 file.

save_to_dict()
Overload this function in your subclass. It should return a dictionary representation of the current instance.

If you member variables that are objects, it is best to convert them to dictionaries before they are entered into your dictionary hierarchy.

Returns **d** – Returns a dictionary representation of the current instance.

Return type dict

class deepdish.util.**NamedRegistry**
This class provides a named hierarchy of classes, where each class is associated with a string name.

classmethod construct(*name*, **args*, ***kwargs*)
Constructs an instance of an object given its name.

classmethod getclass(*name*)
Returns the class object given its name.

name
Returns the name of the registry entry.

classmethod register(*name*)
Decorator to register a class.

classmethod root(*reg_cls*)
Decorate your base class with this, to create a new registry for it

class deepdish.util.**SaveableRegistry**
This combines the features of *deepdish.util.Saveable* and *deepdish.util.NamedRegistry*.

See also:

Saveable, *NamedRegistry*

2.4 Images (deepdish.image)

Basic functions for working with images.

deepdish.image.**asgray**(*im*)
Takes an image and returns its grayscale version by averaging the color channels. if an alpha channel is present, it will simply be ignored. If a grayscale image is given, the original image is returned.

Parameters **image** (*ndarray, ndim 2 or 3*) – RGB or grayscale image.

Returns **gray_image** – Grayscale version of image.

Return type ndarray, ndim 2

deepdish.image.**bounding_box**(*alpha*, *threshold=0.1*)
Returns a bounding box of the support.

Parameters

- **alpha** (*ndarray, ndim=2*) – Any one-channel image where the background has zero or low intensity.
- **threshold** (*float*) – The threshold that divides background from foreground.

Returns **bounding_box** – The bounding box describing the smallest rectangle containing the foreground object, as defined by the threshold.

Return type (top, left, bottom, right)

deepdish.image.**bounding_box_as_binary_map**(*alpha*, *threshold=0.1*)
Similar to *bounding_box*, except returns the bounding box as a binary map the same size as the input.

Same parameters as *bounding_box*.

Returns **binary_map** – Binary map with True if object and False if background.

Return type ndarray, ndim=2, dtype=np.bool_

deepdish.image.**crop**(*im*, *size*)
Crops an image in the center.

Parameters **size** (*tuple, (height, width)*) – Finally size after cropping.

deepdish.image.**crop_or_pad**(*im*, *size*, *value=0*)
Crops an image in the center.

Parameters **size** (*tuple, (height, width)*) – Finally size after cropping.

deepdish.image.**crop_to_bounding_box**(*im*, *bb*)
Crops according to a bounding box.

Parameters **bounding_box** (*tuple, (top, left, bottom, right)*) – Crops inclusively for top/left and exclusively for bottom/right.

deepdish.image.**extract_patches**(*images*, *patch_shape*, *samples_per_image=40*, *seed=0*, *cycle=True*)
Takes a set of images and yields randomly chosen patches of specified size.

Parameters

- **images** (*iterable*) – The images have to be iterable, and each element must be a Numpy array with at least two spatial 2 dimensions as the first and second axis.

- **patch_shape** (*tuple, length 2*) – The spatial shape of the patches that should be extracted. If the images have further dimensions beyond the spatial, the patches will copy these too.
- **samples_per_image** (*int*) – Samples to extract before moving on to the next image.
- **seed** (*int*) – Seed with which to select the patches.
- **cycle** (*bool*) – If True, then the function will produce patches indefinitely, by going back to the first image when all are done. If False, the iteration will stop when there are no more images.

Returns This function returns a generator that will produce patches.

Return type patch_generator

Examples

```
>>> import deepdish as dd
>>> import matplotlib.pylab as plt
>>> import itertools
>>> images = ag.io.load_example('mnist')
```

Now, let us say we want to exact patches from the these, where each patch has at least some activity.

```
>>> gen = dd.image.extract_patches(images, (5, 5))
>>> gen = (x for x in gen if x.mean() > 0.1)
>>> patches = np.array(list(itertools.islice(gen, 25)))
>>> patches.shape
(25, 5, 5)
>>> dd.plot.images(patches)
>>> plt.show()
```

deepdish.image.**integrate**(*ii*, *r0*, *c0*, *r1*, *c1*)

Use an integral image to integrate over a given window.

Parameters

- **ii** (*ndarray*) – Integral image.
- **c0** (*r0,*) – Top-left corner of block to be summed.
- **c1** (*r1,*) – Bottom-right corner of block to be summed.

Returns **S** – Integral (sum) over the given window.

Return type int

deepdish.image.**load**(*path*, *dtype=<Mock id='140083039363280'>*)

Loads an image from file.

Parameters

- **path** (*str*) – Path to image file.
- **dtype** (*np.dtype*) – Defaults to `np.float64`, which means the image will be returned as a float with values between 0 and 1. If `np.uint8` is specified, the values will be between 0 and 255 and no conversion cost will be incurred.

deepdish.image.**offset**(*img*, *offset*, *fill_value=0*)

Moves the contents of image without changing the image size. The missing values are given a specified fill value.

Parameters

- **img** (*array*) – Image.
- **offset** (*(vertical_offset, horizontal_offset)*) – Tuple of length 2, specifying the offset along the two axes.
- **fill_value** (*dtype of img*) – Fill value. Defaults to 0.

deepdish.image.**resize_by_factor**(*im*, *factor*)

Resizes the image according to a factor. The image is pre-filtered with a Gaussian and then resampled with bilinear interpolation.

This function uses scikit-image and essentially combines its *pyramid_reduce* with *pyramid_expand* into one function.

Returns the same object if factor is 1, not a copy.

Parameters

- **im** (*ndarray, ndim=2 or 3*) – Image. Either 2D or 3D with 3 or 4 channels.
- **factor** (*float*) – Resize factor, e.g. a factor of 0.5 will halve both sides.

deepdish.image.**save**(*path*, *im*)

Saves an image to file.

If the image is type float, it will assume to have values in [0, 1].

Parameters

- **path** (*str*) – Path to which the image will be saved.
- **im** (*ndarray (image)*) – Image.

2.5 Parallelization (deepdish.parallel)

See the *Parallelization* chapter for a tutorial.

deepdish.parallel.**rank**()

Returns MPI rank. If the MPI backend is not used, it will always return 0.

deepdish.parallel.**imap_unordered**(*f*, *params*)

This can return the elements in any particular order. This has a lower memory footprint than the ordered version and will be more responsive in terms of printing the results. For instance, if you run the ordered version, and the first batch is particularly slow, you won't see any feedback for a long time.

deepdish.parallel.**imap**(*f*, *params*)

Analogous to *itertools.imap* (Python 2) and *map* (Python 3), but run in parallel.

deepdish.parallel.**starmap_unordered**(*f*, *params*)

Similar to *imap_unordered*, but it will unpack the parameters. That is, it will call `f(*p)`, for each *p* in *params*.

deepdish.parallel.**starmap**(*f*, *params*)

Analogous to *itertools.starmap*, but run in parallel.

deepdish.parallel.**main**(*name=None*)

Main function.

Example use:

```
>>> if gv.parallel.main(__name__):
...     res = gv.parallel.imap_unordered(f, params)
```

CHAPTER 3

Indices and tables

- genindex
- modindex

d