Fuel Documentation

Release 0.1.1

Université de Montréal

January 26, 2016

CHAPTER 1

Installation

The easiest way to install Fuel is using the Python package manager pip. Fuel isn't listed yet on the Python Package Index (PyPI), so you will have to grab it directly from GitHub.

```
$ pip install git+git://github.com/mila-udem/fuel.git
```

This will give you the cutting-edge development version. The latest stable release is in the stable branch and can be installed as follows.

```
$ pip install git+git://github.com/mila-udem/fuel.git@stable
```

If you don't have administrative rights, add the `--user` switch to the install commands to install the packages in your home folder. If you want to update Fuel, simply repeat the first command with the `--upgrade` switch added to pull the latest version from GitHub.

Warning: Pip may try to install or update NumPy and SciPy if they are not present or outdated. However, pip's versions might not be linked to an optimized BLAS implementation. To prevent this from happening make sure you update NumPy and SciPy using your system's package manager (e.g. `apt-get` or `yum`), or use a Python distribution like Anaconda, before installing Fuel. You can also pass the `--no-deps` switch and install all the requirements manually.
If the installation crashes with `ImportError: No module named numpy.distutils.core`, install NumPy and try again again.

1.1 Requirements

Fuel's requirements are

- PyYAML, to parse the configuration file
- six, to support both Python 2 and 3 with a single codebase
- h5py and PyTables for the HDF5 storage back-end
- pillow, providing PIL for image preprocessing
- Cython, for fast extensions
- pyzmq, to efficiently send data across processes
- picklable_itertools, for supporting iterator serialization
- SciPy, to read from MATLAB's .mat format
- requests, to download canonical datasets

nose2 is an optional requirement, used to run the tests.

1.2 Development

If you want to work on Fuel's development, your first step is to fork Fuel on GitHub. You will now want to install your fork of Fuel in editable mode. To install in your home directory, use the following command, replacing `USER` with your own GitHub user name:

```
$ pip install -e git+git@github.com:USER/fuel.git#egg=fuel[test,docs] --src=$HOME
```

As with the usual installation, you can use `--user` or `--no-deps` if you need to. You can now make changes in the `fuel` directory created by pip, push to your repository and make a pull request.

If you had already cloned the GitHub repository, you can use the following command from the folder you cloned Fuel to:

```
$ pip install -e file:.#egg=fuel[test,docs]
```

Fuel contains Cython extensions, which need to be recompiled if you update the Cython *.pyx* files. Each time these files are modified, you should run:

```
$ python setup.py build_ext --inplace
```

1.2.1 Documentation

If you want to build a local copy of the documentation, you can follow the instructions in the documentation development guidelines.

CHAPTER 2

Overview

We'll go over a quick example to see what Fuel is capable of.

Let's start by creating some random data to act as features and targets. We'll pretend that we have eight 2x2 grayscale images separated into four classes.

```
>>> import numpy
>>> seed = 1234
>>> rng = numpy.random.RandomState(seed)
>>> features = rng.randint(256, size=(8, 2, 2))
>>> targets = rng.randint(4, size=(8, 1))
```

Our goal is to use Fuel to interface with this data, iterate over it in various ways and apply transformations to it on the fly.

2.1 Division of labour

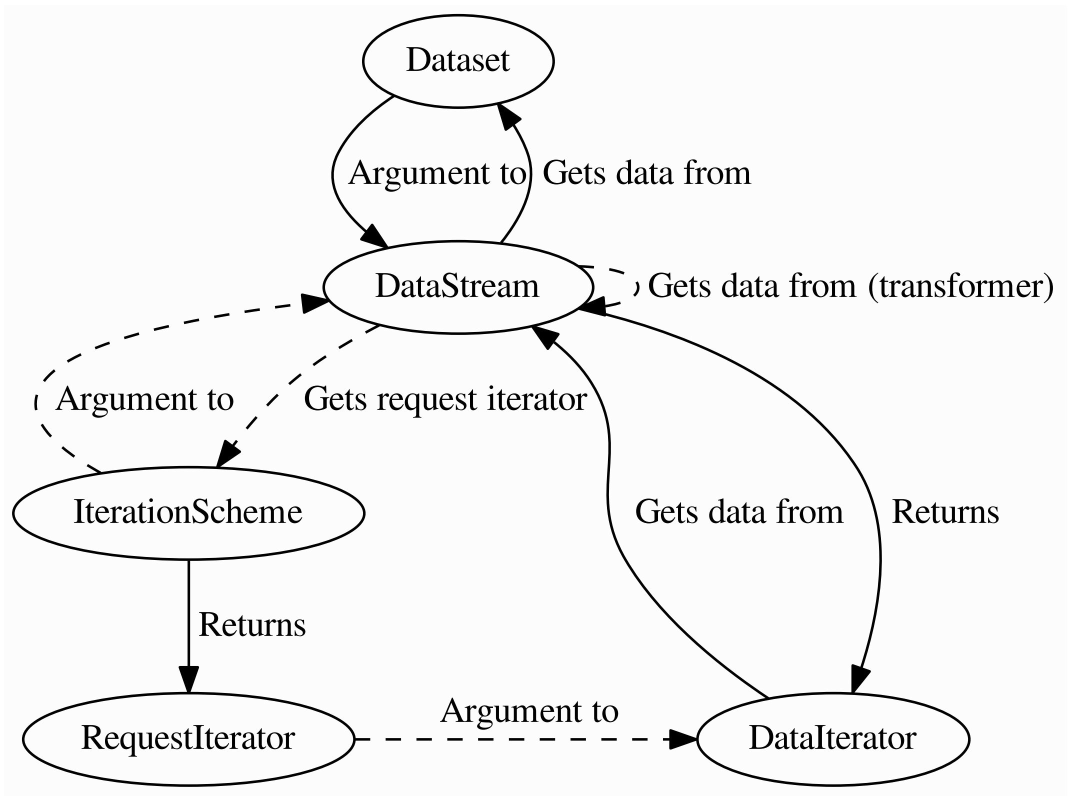
There are four basic tasks that Fuel needs to handle:

- Interface with the data, be it on disk or in memory.
- Decide which data points to visit, and in which order.
- Iterate over the selected data points.
- At each iteration step, apply some transformation to the selected data points.

Each of those four tasks is delegated to a particular class of objects, which we'll be introducing in order.

2.2 Schematic overview of Fuel

For the more visual people, here's a schematic view of how the different components of Fuel interact together. Dashed lines are optional.



2.3 Datasets: interfacing with data

In summary

- `Dataset`
  - Abstract class. Its subclasses are responsible for interfacing with your data.
  - **Constructor arguments**:
    * `sources`: optional, use to restrict which data sources are returned on data requests.
    * `axis_labels`: optional, use to document axis semantics.
  - **Instance attributes**:
    * `sources`: tuple of strings indicating which sources are provided by the dataset, and their ordering (which determines the return order of `get_data()`).
    * `provides_sources`: tuple of source names indicating what sources the dataset *is able to* provide.
    * `axis_labels`: `dict` mapping from source names to tuples of strings or `None`. Used to document the axis semantics of the dataset's sources.
    * `num_examples`: when implemented, represents the number of examples the dataset holds.

  - **Methods used to request data**:
    * `open()`: returns a `state` object the dataset will interact with (e.g. a file handle), or `None` if it doesn't need to interact with anything.
    * `get_data()`: given the `state` object and an optional `request` argument, returns data.
    * `close()`: given the `state` object, properly closes it.
    * `reset()`: given the `state` object, properly closes it and returns a fresh one.

- `IterableDataset`
  - Allows to interface with iterable objects.
  - The state `IterableDataset.open()` returns is an iterator object.
  - Its `get_data()` method doesn't accept requests.
  - Can only iterate examplewise and sequentially.
  - **Constructor arguments**:
    * `iterables`: a `dict` mapping from source names to their corresponding iterable objects. Use `collections.OrderedDict` instances if the source order is important to you.
- `IndexableDataset`
  - Allows to interface with indexable objects.
  - The state `IndexableDataset.open()` returns is `None`.
  - Its `get_data()` method accepts requests.
  - Allows random access.
  - **Constructor arguments**:
    * `indexables`: a `dict` mapping from source names to their corresponding indexable objects. Use `collections.OrderedDict` instances if the source order is important to you.

The `Dataset` class is responsible for interfacing with the data and handling data access requests. Subclasses of `Dataset` specialize in certain types of data.

Datasets contain one or more **sources** of data, such as an array of images, a list of labels, a dictionary specifying an ontology, etc. Each source in a dataset is identified by a unique name.

All datasets have the following attributes:

- `sources`: tuple of source names indicating what the dataset will provide when queried for data.
- `provides_sources`: tuple of source names indicating what sources the dataset *is able to* provide.
- `axis_labels`: `dict` mapping each source name to a tuple of axis labels, or `None`. Not all source names need to appear in the axis labels dictionary.

Some datasets also have a `num_examples` attribute telling how many examples the dataset provides.

2.3.1 IterableDataset

The simplest `Dataset` subclass is `IterableDataset`, which interfaces with iterable objects.

It is created by passing an `iterables` `dict` mapping source names to their associated data and, optionally, an `axis_labels` `dict` mapping source names to their corresponding tuple of axis labels.

```
>>> from collections import OrderedDict
>>> from fuel.datasets import IterableDataset
>>> dataset = IterableDataset(
...     iterables=OrderedDict([('features', features), ('targets', targets)]),
...     axis_labels=OrderedDict([('features', ('batch', 'height', 'width')),
...                              ('targets', ('batch', 'index'))]))
```

We can access the `sources`, `provides_sources` and `axis_labels` attributes defined in all datasets, as well as `num_examples`.

```
>>> print('Provided sources are {}.'.format(dataset.provides_sources))
Provided sources are ('features', 'targets').
>>> print('Sources are {}.'.format(dataset.sources))
Sources are ('features', 'targets').
>>> print('Axis labels are {}.'.format(dataset.axis_labels))
Axis labels are OrderedDict([('features', ('batch', 'height', 'width')), ('targets', ('batch', 'index
>>> print('Dataset contains {} examples.'.format(dataset.num_examples))
Dataset contains 8 examples.
```

Tip: The source order of an `IterableDataset` instance depends on the key order of `iterables`, which is non-deterministic for regular `dict` instances. We therefore recommend that you use `collections.OrderedDict` instances if the source order is important to you.

Datasets themselves are stateless objects (as opposed to, say, an open file handle, or an iterator object). In order to request data from the dataset, we need to ask it to instantiate some stateful object with which it will interact. This is done through the `Dataset.open()` method:

```
>>> state = dataset.open()
>>> print(state.__class__.__name__)
imap
```

We can see that in `IterableDataset`'s case the state is an iterator (`imap`) object. We can now visit the examples this dataset contains using its `get_data()` method.

```
>>> while True:
...     try:
...         print(dataset.get_data(state=state))
...     except StopIteration:
...         print('Iteration over')
...         break
(array([[ 47, 211],
       [ 38,  53]]), array([0]))
(array([[204, 116],
       [152, 249]]), array([3]))
(array([[143, 177],
       [ 23, 233]]), array([0]))
(array([[154,  30],
       [171, 158]]), array([1]))
(array([[236, 124],
       [ 26, 118]]), array([2]))
(array([[186, 120],
       [112, 220]]), array([2]))
(array([[ 69,  80],
       [201, 127]]), array([2]))
(array([[246, 254],
       [175,  50]]), array([3]))
Iteration over
```

Eventually, the iterator is depleted and it raises a `StopIteration` exception. We can iterate over the dataset again by requesting a fresh iterator through the dataset's `reset()` method.

```
>>> state = dataset.reset(state=state)
>>> print(dataset.get_data(state=state))
(array([[ 47, 211],
       [ 38,  53]]), array([0]))
```

When you're done, don't forget to call the dataset's `close()` method on the state. This has the effect of cleanly closing the state (e.g. if the state is an open file handle, `close()` will close it).

```
>>> dataset.close(state=state)
```

2.3.2 IndexableDataset

The `IterableDataset` implementation is pretty minimal. For instance, it only lets you iterate sequentially and examplewise over your data.

If your data happens to be indexable (e.g. a `list`, or a `numpy.ndarray`), then `IndexableDataset` will let you do much more.

We instantiate `IndexableDataset` just like `IterableDataset`.

```
>>> from fuel.datasets import IndexableDataset
>>> dataset = IndexableDataset(
...     indexables=OrderedDict([('features', features), ('targets', targets)]),
...     axis_labels=OrderedDict([('features', ('batch', 'height', 'width')),
...                              ('targets', ('batch', 'index'))]))
```

The main advantage of `IndexableDataset` over `IterableDataset` is that it allows random access of the data it contains. In order to do so, we need to pass an additional `request` argument to `get_data()` in the form of a list of indices.

```
>>> state = dataset.open()
>>> print('State is {}.'.format(state))
State is None.
>>> print(dataset.get_data(state=state, request=[0, 1]))
(array([[[ 47, 211],
        [ 38,  53]],

       [[204, 116],
        [152, 249]]]), array([[0],
       [3]]))
>>> dataset.close(state=state)
```

See how `IndexableDataset` returns a `None` state: this is because there's no actual state to maintain in this case.

2.3.3 Restricting sources

In some cases (e.g. unsupervised learning), you might want to use a subset of the provided sources. This is achieved by passing a `sources` argument to the dataset constructor. Here's an example:

```
>>> restricted_dataset = IndexableDataset(
...     indexables=OrderedDict([('features', features), ('targets', targets)]),
...     axis_labels=OrderedDict([('features', ('batch', 'height', 'width')),
...                              ('targets', ('batch', 'index'))]),
...     sources=('features',))
```

```
>>> print(restricted_dataset.provides_sources)
('features', 'targets')
>>> print(restricted_dataset.sources)
('features',)
>>> state = restricted_dataset.open()
>>> print(restricted_dataset.get_data(state=state, request=[0, 1]))
(array([[[ 47, 211],
        [ 38,  53]],

       [[204, 116],
        [152, 249]]]),)
>>> restricted_dataset.close(state=state)
```

You can see that in this case only the features are returned by `get_data()`.

2.4 Iteration schemes: which examples to visit

In summary

- `IterationScheme`
  - Abstract class. Its subclasses are responsible for deciding in which order examples are visited.
  - **Methods**:
    * `get_request_iterator()`: returns an iterator object that returns requests. These requests can be fed to a dataset's `get_data()` method.
- `BatchScheme`
  - Abstract class. Its subclasses return batch requests.
  - Commonly used subclasses are:
    * `SequentialScheme`: requests batches sequentially.
    * `ShuffledScheme`: requests batches in shuffled order.
- `IndexScheme`
  - Abstract class. Its subclasses return example requests.
  - Commonly used subclasses are:
    * `SequentialExampleScheme`: requests examples sequentially.
    * `ShuffledExampleScheme`: requests examples in shuffled order.

Encapsulating and accessing our data is good, but if we're to integrate it into a training loop, we need to be able to iterate over the data. For that, we need to decide *which* indices to request and in *which order*. This is accomplished via an `IterationScheme` subclass.

At its most basic level, an iteration scheme is responsible, through its `get_request_iterator()` method, for building an iterator that will return requests. Here are some examples:

```
>>> from fuel.schemes import (SequentialScheme, ShuffledScheme,
...                           SequentialExampleScheme, ShuffledExampleScheme)
>>> schemes = [SequentialScheme(examples=8, batch_size=4),
...            ShuffledScheme(examples=8, batch_size=4),
...            SequentialExampleScheme(examples=8),
...             ShuffledExampleScheme(examples=8)]
>>> for scheme in schemes:
...     print(list(scheme.get_request_iterator()))
[[0, 1, 2, 3], [4, 5, 6, 7]]
[[7, 2, 1, 6], [0, 4, 3, 5]]
[0, 1, 2, 3, 4, 5, 6, 7]
[7, 2, 1, 6, 0, 4, 3, 5]
```

We can therefore use an iteration scheme to visit a dataset in some order.

```
>>> state = dataset.open()
>>> scheme = ShuffledScheme(examples=dataset.num_examples, batch_size=4)
>>> for request in scheme.get_request_iterator():
...     data = dataset.get_data(state=state, request=request)
...     print(data[0].shape, data[1].shape)
(4, 2, 2) (4, 1)
(4, 2, 2) (4, 1)
>>> dataset.close(state)
```

Note: Not all iteration schemes work with all datasets. For instance, `IterableDataset` doesn't work with any iteration scheme, since its `get_data()` method doesn't accept requests.

2.5 Data streams: automating the iteration process

In summary

- `AbstractDataStream`
  - Abstract class. Its subclasses are responsible for coordinating a dataset and an iteration scheme to iterate through the data.
  - **Methods for iterating**:
    - `get_epoch_iterator()`: returns an iterator that returns examples or batches of examples.
  - **Constructor arguments**:
    - `iteration_scheme`: `IterationScheme` instance, optional, use to specify the iteration order.
    - `axis_labels`: optional, use to document axis semantics.
- `DataStream`
  - The most common data stream.
  - **Constructor arguments**:
    - `dataset`: `Dataset` instance, which dataset to iterate over.

Iteration schemes offer a more convenient way to visit the dataset than accessing the data by hand, but we can do better: the act of getting a fresh state from the dataset, getting a request iterator from the iteration scheme, using both to access the data and closing the state is repetitive. To automate this, we have *data streams*, which are subclasses of `AbstractDataStream`.

The most common `AbstractDataStream` subclass is `DataStream`. It is instantiated with a dataset and an iteration scheme, and returns an epoch iterator through its `get_epoch_iterator()` method, which iterates over the dataset in the order defined by the iteration scheme.

```
>>> from fuel.streams import DataStream
>>> data_stream = DataStream(dataset=dataset, iteration_scheme=scheme)
>>> for data in data_stream.get_epoch_iterator():
...     print(data[0].shape, data[1].shape)
(4, 2, 2) (4, 1)
(4, 2, 2) (4, 1)
```

2.6 Transformers: apply some transformation on the fly

In summary

- Transformer
  - AbstractDataStream subclass. Is itself an abstract class. Its subclasses are responsible for taking data stream(s) as input and producing a data stream as output, which applies some transformation to the input stream(s).
  - Transformers can be chained together to form complex data processing pipelines.
  - **Constructor arguments**:
    * data_stream: AbstractDataStream instance, the input stream.

Some AbstractDataStream subclasses take data streams as input. We call them *transformers*, and they enable us to build complex data preprocessing pipelines.

Transformers are Transformer subclasses, which is itself an AbstractDataStream subclass. Here are some commonly used ones:

- Flatten: flattens the input into a matrix (for batch input) or a vector (for examplewise input).
- ScaleAndShift: scales and shifts the input by scalar quantities.
- Cast: casts the input into some data type.

As an example, let's standardize the images we have by substracting their mean and dividing by their standard deviation.

```
>>> from fuel.transformers import ScaleAndShift
>>> # Note: ScaleAndShift applies (batch * scale) + shift, as
>>> # opposed to (batch + shift) * scale.
>>> scale = 1.0 / features.std()
>>> shift = - scale * features.mean()
>>> standardized_stream = ScaleAndShift(data_stream=data_stream,
...                                     scale=scale, shift=shift,
...                                     which_sources=('features',))
```

The resulting data stream can be used to iterate over the dataset just like before, but this time features will be standardized on-the-fly.

```
>>> for batch in standardized_stream.get_epoch_iterator():
...     print(batch)
(array([[[ 0.18530572, -1.54479571],
        [ 0.42249705,  0.24111545]],

       [[-1.30760439,  0.98059429],
        [-1.43317627, -1.2238898 ]],
       [[ 1.46892937,  1.58054882],
        [ 0.47830677, -1.2657471 ]],

       [[ 0.63178351, -0.28907693],
        [-0.40069638,  1.10616617]]]), array([[1],
       [0],
       [3],
       [2]]))
(array([[[ 1.32940506, -0.2332672 ],
        [-1.60060544, -0.31698179]],

       [[ 0.03182898,  0.50621164],
        [-1.64246273,  1.28754777]],

       [[ 0.88292727, -0.34488665],
        [ 0.15740086,  1.51078666]],

       [[-1.00065091, -0.84717417],
        [ 0.84106998, -0.19140991]]]), array([[2],
       [0],
       [3],
       [2]]))
```

Now, let's imagine that for some reason (e.g. running Theano code on GPU) we need features to have a data type of `float32`. We can cast them on-the-fly with a `Cast` transformer.

```
>>> from fuel.transformers import Cast
>>> cast_standardized_stream = Cast(
...     data_stream=standardized_stream,
...     dtype='float32', which_sources=('features',))
```

As you can see, Fuel makes it easy to chain transformations to form a preprocessing pipeline. The complete pipeline now looks like this:

```
>>> data_stream = Cast(
...     ScaleAndShift(
...         DataStream(
...             dataset=dataset, iteration_scheme=scheme),
...         scale=scale, shift=shift, which_sources=('features',)),
...     dtype='float32', which_sources=('features',))
```

2.7 Going further

You now know enough to find your way around Fuel. Here are the next steps:

- Learn how to use built-in datasets.
- Learn how to import your own data in Fuel.
- Learn how to extend Fuel to suit your needs.

CHAPTER 3

Built-in datasets

Fuel has a growing number of built-in datasets that simplify working on standard benchmark datasets, such as MNIST or CIFAR10.

These datasets are defined in the `fuel.datasets` module. Some user intervention is needed before they're used for the first time: a given dataset has to be downloaded and converted into a format that is recognized by its corresponding dataset class. Fortunately, Fuel also has built-in tools to automate these operations.

3.1 Environment variable

In order for Fuel to know where to look for its data, the `data_path` configuration variable has to be set inside `~/.fuelrc`. It's expected to be a sequence of paths separated by an OS-specific delimiter (`:` for Linux and OSX, `;` for Windows):

```
# ~/.fuelrc
data_path: "/first/path/to/my/data:/second/path/to/my/data"
```

When looking for a specific file (e.g. `mnist.hdf5`), Fuel will search each of these paths in sequence, using the first matching file that it finds.

This configuration variable can be overridden by setting the `FUEL_DATA_PATH` environment variable:

```
$ export FUEL_DATA_PATH="/first/path/to/my/data:/second/path/to/my/data"
```

Let's now change directory for the rest of this tutorial:

```
$ cd $FUEL_DATA_PATH
```

3.2 Download a built-in dataset

We're going to download the raw data files for the MNIST dataset with the `fuel-download` script that was installed with Fuel:

```
$ fuel-download mnist
```

The script is pretty simple: you call it and pass it the name of the dataset you'd like to download. In order to know which datasets are available to download via `fuel-download`, type

```
$ fuel-download -h
```

You can pass dataset-specific arguments to the script. In order to know which arguments are accepted, append `-h` to your dataset choice:

```
fuel-download mnist -h
```

Two arguments are always accepted:

- `-d DIRECTORY` : define where the dataset files will be downloaded. By default, `fuel-download` uses the current working directory.
- `--clear` : delete the dataset files instead of downloading them, if they exist.

3.3 Convert downloaded files

You should now have four new files in your directory:

- `train-images-idx3-ubyte.gz`
- `train-labels-idx1-ubyte.gz`
- `t10k-images-idx3-ubyte.gz`
- `t10k-labels-idx1-ubyte.gz`

Those are the original files that can be downloaded off Yann Lecun's website. We now need to convert those files into a format that the `MNIST` dataset class will recognize. This is done through the `fuel-convert` script:

```
$ fuel-convert mnist
```

This will generate an `mnist.hdf5` file in your directory, which the `MNIST` class recognizes.

Once again, the script accepts dataset-specific arguments which you can discover by appending `-h` to your dataset choice:

```
fuel-convert mnist -h
```

Two arguments are always accepted:

- `-d DIRECTORY` : where `fuel-convert` should look for the input files.
- `-o OUTPUT_FILE` : where to save the converted dataset.

Let's delete the raw input files, as we don't need them anymore:

```
$ fuel-download mnist --clear
```

3.4 Inspect Fuel-generated dataset files

Six months from now, you may have a bunch of dataset files lying on disk, each with slight differences that you can't identify or reproduce. At that time, you'll be glad that `fuel-info` exists.

When a dataset is generated through `fuel-convert`, the script tags it with what command was issued to generate the file and what were the versions of relevant parts of the library at that time.

You can inspect this metadata calling `fuel-info` and passing an HDF5 file as argument:

```
$ fuel-info mnist.hdf5
```

```
Metadata for mnist.hdf5
=======================

The command used to generate this file is

    fuel-convert mnist

Relevant versions are

    H5PYDataset     0.1
    fuel.converters 0.1
```

3.5 Working with external packages

By default, Fuel looks for downloaders and converters in the `fuel.downloaders` and `fuel.converters` modules, respectively, but you're not limited to that.

Fuel can be told to look into additional modules by setting the `extra_downloaders` and `extra_converters` configuration variables in `~/.fuelrc`. These variables are expected to be lists of module names.

For instance, suppose you'd like to include the following modules:

- `package1.extra_downloaders`
- `package2.extra_downloaders`
- `package1.extra_converters`
- `package2.extra_converters`

You should include the following in your `~/.fuelrc`:

```
# ~/.fuelrc
extra_downloaders:
- package1.extra_downloaders
- package2.extra_downloaders
extra_converters:
- package1.extra_converters
- package2.extra_converters
```

These configuration variables can be overridden through the `FUEL_EXTRA_DOWNLOADERS` and `FUEL_EXTRA_CONVERTERS` environment variables, which are expected to be strings of space-separated module names, like so:

```
export FUEL_EXTRA_DOWNLOADERS="package1.extra_downloaders package2.extra_downloaders"
export FUEL_EXTRA_CONVERTERS="package1.extra_converters package2.extra_converters"
```

This feature lets external developers define their own Fuel dataset downloader/converter packages, and also makes working with private datasets more straightforward.

CHAPTER 4

Getting your data in Fuel

Warning: We're still in the process of figuring out the interface, which means the "preferred" way of getting your data in Fuel may change in the future.

Built-in datasets are convenient for training on common benchmark tasks, but what if you want to train a model on your own data?

This section shows how to accomplish the common task of loading into Fuel a bunch of data sources (*e.g.* features, targets) split into different sets.

4.1 A toy example

We'll start this tutorial by creating bogus data sources that could be lying around on disk.

```
>>> import numpy
>>> numpy.save(
...     'train_vector_features.npy',
...     numpy.random.normal(size=(90, 10)).astype('float32'))
>>> numpy.save(
...     'test_vector_features.npy',
...     numpy.random.normal(size=(10, 10)).astype('float32'))
>>> numpy.save(
...     'train_image_features.npy',
...     numpy.random.randint(2, size=(90, 3, 5, 5)).astype('uint8'))
>>> numpy.save(
...     'test_image_features.npy',
...     numpy.random.randint(2, size=(10, 3, 5, 5)).astype('uint8'))
>>> numpy.save(
...     'train_targets.npy',
...     numpy.random.randint(10, size=(90, 1)).astype('uint8'))
>>> numpy.save(
...     'test_targets.npy',
...     numpy.random.randint(10, size=(10, 1)).astype('uint8'))
```

Our goal is to process these files into a format that can be natively imported in Fuel.

4.2 HDF5 datasets

The best-supported way to load data in Fuel is through the `H5PYDataset` class, which wraps HDF5 files using `h5py`.

This is the class that's used for most built-in datasets. It makes a series of assumptions about the structure of the HDF5 file which greatly simplify things if your data happens to meet these assumptions:

- All data is stored into a single HDF5 file.
- Data sources reside in the root group, and their names define the source names.
- Data sources are not explicitly split into separate HDF5 datasets or separate HDF5 files. Instead, splits are defined in the `split` attribute of the root group. It's expected to be a 1D numpy array of compound `dtype` with seven fields, organized as follows:
    1. `split` : string identifier for the split name
    2. `source` : string identifier for the source name
    3. `start` : start index (inclusive) of the split in the source array, used if `indices` is a null reference.
    4. `stop` : stop index (exclusive) of the split in the source array, used if `indices` is a null reference.
    5. `indices` : h5py.Reference, reference to a dataset containing subset indices for this split/source pair. If it's a null reference, `start` and `stop` are used.
    6. `available` : boolean, `False` is this split is not available for this source
    7. `comment` : comment string

Tip: Some of you may wonder if this means all data has to be read off disk all the time. Rest assured, `H5PYDataset` has an option to load things into memory which we will be covering soon.

4.3 Converting the toy example

Let's now convert our bogus files into a format that's digestible by `H5PYDataset`.

We first load the data from disk.

```
>>> train_vector_features = numpy.load('train_vector_features.npy')
>>> test_vector_features = numpy.load('test_vector_features.npy')
>>> train_image_features = numpy.load('train_image_features.npy')
>>> test_image_features = numpy.load('test_image_features.npy')
>>> train_targets = numpy.load('train_targets.npy')
>>> test_targets = numpy.load('test_targets.npy')
```

We then open an HDF5 file for writing and create three datasets in the root group, one for each data source. We name them after their source name.

```
>>> import h5py
>>> f = h5py.File('dataset.hdf5', mode='w')
>>> vector_features = f.create_dataset(
...     'vector_features', (100, 10), dtype='float32')
>>> image_features = f.create_dataset(
...     'image_features', (100, 3, 5, 5), dtype='uint8')
>>> targets = f.create_dataset(
...     'targets', (100, 1), dtype='uint8')
```

Notice how the number of examples we specify (100) in the shapes is the sum of the number of training and test examples. We'll be filling the first 90 rows with training examples and the last 10 rows with test examples.

```
>>> vector_features[...] = numpy.vstack(
...     [train_vector_features, test_vector_features])
>>> image_features[...] = numpy.vstack(
...     [train_image_features, test_image_features])
>>> targets[...] = numpy.vstack([train_targets, test_targets])
```

`H5PYDataset` allows us to label axes with semantic information. We record that information in the HDF5 file through dimension scales.

```
>>> vector_features.dims[0].label = 'batch'
>>> vector_features.dims[1].label = 'feature'
>>> image_features.dims[0].label = 'batch'
>>> image_features.dims[1].label = 'channel'
>>> image_features.dims[2].label = 'height'
>>> image_features.dims[3].label = 'width'
>>> targets.dims[0].label = 'batch'
>>> targets.dims[1].label = 'index'
```

This particular choice of label is arbitrary. Nothing in Fuel forces you to adopt any labeling convention. Note, however, that certain external frameworks that rely on Fuel *may* impose some restrictions on the choice of labels.

The last thing we need to do is to give `H5PYDataset` a way to recover what the splits are. This is done by setting the `split` attribute of the root group.

```
>>> split_array = numpy.empty(
...     6,
...     dtype=numpy.dtype([
...         ('split', 'a', 5),
...         ('source', 'a', 15),
...         ('start', numpy.int64, 1),
...         ('stop', numpy.int64, 1),
...         ('indices', h5py.special_dtype(ref=h5py.Reference)),
...         ('available', numpy.bool, 1),
...         ('comment', 'a', 1)]))
>>> split_array[0:3]['split'] = 'train'.encode('utf8')
>>> split_array[3:6]['split'] = 'test'.encode('utf8')
>>> split_array[0:6:3]['source'] = 'vector_features'.encode('utf8')
>>> split_array[1:6:3]['source'] = 'image_features'.encode('utf8')
>>> split_array[2:6:3]['source'] = 'targets'.encode('utf8')
>>> split_array[0:3]['start'] = 0
>>> split_array[0:3]['stop'] = 90
>>> split_array[3:6]['start'] = 90
>>> split_array[3:6]['stop'] = 100
>>> split_array[:]['indices'] = h5py.Reference()
>>> split_array[:]['available'] = True
>>> split_array[:]['comment'] = '.'.encode('utf8')
>>> f.attrs['split'] = split_array
```

We created a 1D numpy array with six elements. The `dtype` for this array is a compound type: every element of the array is a tuple of `(str, str, int, int, h5py.Reference, bool, str)`. The length of each string element has been chosen to be the maximum length we needed to store: that's 5 for the `split` element (`'train'` being the longest split name) and 15 for the `source` element (`'vector_features'` being the longest source name). We didn't include any comment, so the length for that element was set to 1. Due to a quirk in pickling empty strings, we put `'.'` as the comment value.

Warning: Due to limitations in h5py, you must make sure to use bytes for `split`, `source` and `comment`.

H5PYDataset expects the split attribute of the root node to contain as many elements as the cartesian product of all sources and all splits, *i.e.* all possible split/source combinations. Sometimes, no data is available for some source/split combination: for instance, the test set may not be labeled, and the ('test', 'targets') combination may not exist. In that case, you can set the available element for that combination to False, and H5PYDataset will ignore it.

Don't worry too much about indices; we'll get back to that later. For the moment, all you need to know is that since our splits are contiguous, we don't need that feature and therefore put empty references.

The method described above does the job, but it's not very convenient. An even simpler way of achieving the same result is to call create_split_array().

```
>>> from fuel.datasets.hdf5 import H5PYDataset
>>> split_dict = {
...     'train': {'vector_features': (0, 90), 'image_features': (0, 90),
...               'targets': (0, 90)},
...     'test': {'vector_features': (90, 100), 'image_features': (90, 100),
...              'targets': (90, 100)}}
>>> f.attrs['split'] = H5PYDataset.create_split_array(split_dict)
```

The create_split_array() method expects a dictionary mapping split names to dictionaries. Those dictionaries map source names to tuples of length 2, 3 or 4. The first two elements correspond to the start and stop indexes. The other two elements are optional and correspond to the indices reference and the comment, respectively. The method will create the array behind the scenes, choose the string lengths automatically and populate it with the information in the split dictionary. If a particular split/source combination isn't present, its available attribute is set to False, which allows us to specify only what's actually present in the HDF5 file we created.

Tip: By default, H5PYDataset sorts sources in alphabetical order, and data requests are also returned in that order. If sources is passed as argument upon instantiation, H5PYDataset will use the order of sources instead. This means that if you want to force a particular source order, you can do so by explicitly passing the sources argument with the desired ordering. For example, if your dataset has two sources named 'features' and 'targets' and you'd like the targets to be returned first, you need to pass sources=('targets', 'features') as a constructor argument.

We flush, close the file and *voilà*!

```
>>> f.flush()
>>> f.close()
```

4.4 Playing with H5PYDataset datasets

Let's explore what we can do with the dataset we just created.

The simplest thing is to load it by giving its path and a tuple of split names:

```
>>> train_set = H5PYDataset('dataset.hdf5', which_sets=('train',))
>>> print(train_set.num_examples)
90
>>> test_set = H5PYDataset('dataset.hdf5', which_sets=('test',))
>>> print(test_set.num_examples)
10
```

Passing more than one split name would cause the splits to be concatenated. The available data sources would be the intersection of the sources provided by each split.

You can further restrict which examples are used by providing a `slice` object or a list of indices as the `subset` argument.

```
>>> train_set = H5PYDataset(
...     'dataset.hdf5', which_sets=('train',), subset=slice(0, 80))
>>> print(train_set.num_examples)
80
>>> valid_set = H5PYDataset(
...     'dataset.hdf5', which_sets=('train',), subset=slice(80, 90))
>>> print(valid_set.num_examples)
10
```

The available data sources are defined by the names of the datasets in the root node of the HDF5 file, and `H5PYDataset` automatically picked them up for us:

```
>>> print(train_set.provides_sources)
('image_features', 'targets', 'vector_features')
```

It also parsed axis labels, which are accessible through the `axis_labels` property, which is a dict mapping source names to a tuple of axis labels:

```
>>> print(train_set.axis_labels['image_features'])
('batch', 'channel', 'height', 'width')
>>> print(train_set.axis_labels['vector_features'])
('batch', 'feature')
>>> print(train_set.axis_labels['targets'])
('batch', 'index')
```

We can request data as usual:

```
>>> handle = train_set.open()
>>> data = train_set.get_data(handle, slice(0, 10))
>>> print((data[0].shape, data[1].shape, data[2].shape))
((10, 3, 5, 5), (10, 1), (10, 10))
>>> train_set.close(handle)
```

We can also request just the vector features:

```
>>> train_vector_features = H5PYDataset(
...     'dataset.hdf5', which_sets=('train',), subset=slice(0, 80),
...     sources=['vector_features'])
>>> handle = train_vector_features.open()
>>> data, = train_vector_features.get_data(handle, slice(0, 10))
>>> print(data.shape)
(10, 10)
>>> train_vector_features.close(handle)
```

4.5 Loading data in memory

Reading data off disk is inefficient compared to storing it in memory. Large datasets make it inevitable, but if your dataset is small enough that it fits into memory, you should take advantage of it.

In `H5PYDataset`, this is accomplished via the `load_in_memory` constructor argument. It has the effect of loading *just* what you requested, and nothing more.

```
>>> in_memory_train_vector_features = H5PYDataset(
...     'dataset.hdf5', which_sets=('train',), subset=slice(0, 80),
...     sources=['vector_features'], load_in_memory=True)
```

```
>>> data, = in_memory_train_vector_features.data_sources
>>> print(type(data))
<... 'numpy.ndarray'>
>>> print(data.shape)
(80, 10)
```

4.6 Non-contiguous splits

Sometimes it's not possible to store the different splits contiguously. In that case, you'll want to use the `indices` field of the `H5PYDataset` split array. A non-empty reference in that field overrides the `start` and `stop` fields, and the dataset the reference points to is used to determine the indices for that split/source pair.

Suppose that you'd like to use the even examples as your training set and the odd examples as your test set. We'll start with the HDF5 file we populated earlier and manipulate its `split` attribute.

```
>>> f = h5py.File('dataset.hdf5', mode='a')
>>> f['train_indices'] = numpy.arange(0, 100, 2)
>>> train_ref = f['train_indices'].ref
>>> f['test_indices'] = numpy.arange(1, 100, 2)
>>> test_ref = f['test_indices'].ref
>>> split_dict = {
...     'train': {'vector_features': (-1, -1, train_ref),
...               'image_features': (-1, -1, train_ref),
...               'targets': (-1, -1, train_ref)},
...     'test': {'vector_features': (-1, -1, test_ref),
...              'image_features': (-1, -1, test_ref),
...              'targets': (-1, -1, test_ref)}}
>>> f.attrs['split'] = H5PYDataset.create_split_array(split_dict)
>>> f.flush()
>>> f.close()
```

We created two new datasets containing even and odd indices from 0 to 99, respectively, and passed references to these datasets in the split dict. In that case, the value we pass to `start` and `stop` really doesn't matter, so we arbitrarily chose `-1` for both.

Let's check that the training and test set do contain even and odd examples:

```
>>> train_set = H5PYDataset(
...     'dataset.hdf5', which_sets=('train',), sources=('vector_features',))
>>> handle = train_set.open()
>>> print(
...     numpy.array_equal(
...         train_set.get_data(handle, slice(0, 50))[0],
...         numpy.vstack(
...             [numpy.load('train_vector_features.npy'),
...              numpy.load('test_vector_features.npy')])[::2]))
True
>>> train_set.close(handle)
>>> test_set = H5PYDataset(
...     'dataset.hdf5', which_sets=('test',), sources=('vector_features',))
>>> handle = test_set.open()
>>> print(
...     numpy.array_equal(
...         test_set.get_data(handle, slice(0, 50))[0],
...         numpy.vstack(
...             [numpy.load('train_vector_features.npy'),
...              numpy.load('test_vector_features.npy')])[1::2]))
```

```
True
>>> test_set.close(handle)
```

4.7 Variable-length data

`H5PYDataset` also supports variable length data. Let's update the image features to reflect that:

```
>>> sizes = numpy.random.randint(3, 9, size=(100,))
>>> train_image_features = [
...     numpy.random.randint(256, size=(3, size, size)).astype('uint8')
...     for size in sizes[:90]]
>>> test_image_features = [
...     numpy.random.randint(256, size=(3, size, size)).astype('uint8')
...     for size in sizes[90:]]
```

In this new example, images have random shapes ranging from 3x3 pixels to 8x8 pixels.

First, we put the vector features and the targets inside the HDF5 file as before:

```
>>> f = h5py.File('dataset.hdf5', mode='w')
>>> f['vector_features'] = numpy.vstack(
...     [numpy.load('train_vector_features.npy'),
...      numpy.load('test_vector_features.npy')])
>>> f['targets'] = numpy.vstack(
...     [numpy.load('train_targets.npy'),
...      numpy.load('test_targets.npy')])
>>> f['vector_features'].dims[0].label = 'batch'
>>> f['vector_features'].dims[1].label = 'feature'
>>> f['targets'].dims[0].label = 'batch'
>>> f['targets'].dims[1].label = 'index'
```

We now have to put the variable-length images inside the HDF5 file. We can't do that directly, since HDF5 and h5py don't support multi-dimensional ragged arrays. However, there *is* support for 1D ragged arrays. Instead, we'll flatten the images before putting them in the HDF5 file:

```
>>> all_image_features = train_image_features + test_image_features
>>> dtype = h5py.special_dtype(vlen=numpy.dtype('uint8'))
>>> image_features = f.create_dataset('image_features', (100,), dtype=dtype)
>>> image_features[...] = [image.flatten() for image in all_image_features]
>>> image_features.dims[0].label = 'batch'
```

If you're feeling lost, have a look at the dedicated tutorial on variable-length data.

The images are now in the HDF5 file, but that doesn't help us unless we can recover their original shape. For that, we'll create a dimension scale that we'll attach to the `'image_features'` dataset using the name `'shapes'` (use this *exact* name):

```
>>> image_features_shapes = f.create_dataset(
...     'image_features_shapes', (100, 3), dtype='int32')
>>> image_features_shapes[...] = numpy.array(
...     [image.shape for image in all_image_features])
>>> image_features.dims.create_scale(image_features_shapes, 'shapes')
>>> image_features.dims[0].attach_scale(image_features_shapes)
```

We'd also like to tag those variable-length dimensions with semantic information. We'll create another dimension scale that we'll attach to the `'image_features'` dataset using the name `'shape_labels'` (use this *exact* name):

```
>>> image_features_shape_labels = f.create_dataset(
...     'image_features_shape_labels', (3,), dtype='S7')
>>> image_features_shape_labels[...] = [
...     'channel'.encode('utf8'), 'height'.encode('utf8'),
...     'width'.encode('utf8')]
>>> image_features.dims.create_scale(
...     image_features_shape_labels, 'shape_labels')
>>> image_features.dims[0].attach_scale(image_features_shape_labels)
```

The H5PYDataset class will handle things from there on. When image features are loaded, it will retrieve their shapes and do the reshape automatically.

Lastly, we create the split dictionary exactly as before:

```
>>> split_dict = {
...     'train': {'vector_features': (0, 90), 'image_features': (0, 90),
...               'targets': (0, 90)},
...     'test': {'vector_features': (90, 100), 'image_features': (90, 100),
...              'targets': (90, 100)}}
>>> f.attrs['split'] = H5PYDataset.create_split_array(split_dict)
>>> f.flush()
>>> f.close()
```

That's it. Now let's kick the tires a little. The axis labels appear as they should:

```
>>> train_set = H5PYDataset(
...     'dataset.hdf5', which_sets=('train',), sources=('image_features',))
>>> print(train_set.axis_labels['image_features'])
('batch', 'channel', 'height', 'width')
```

H5PYDataset retrieves images of different shapes and automatically unflattens them:

```
>>> handle = train_set.open()
>>> images, = train_set.get_data(handle, slice(0, 10))
>>> train_set.close(handle)
>>> print(images[0].shape, images[1].shape)
(3, 6, 6) (3, 8, 8)
```

The object returned by get_data is a 1D numpy array of objects:

```
>>> print(type(images), images.dtype, images.shape)
<... 'numpy.ndarray'> object (10,)
```

Contributing a dataset to Fuel

This tutorial describes what you need to implement in order to contribute a new dataset to Fuel.

You need to implement the following:

- Code that downloads the raw data files for your dataset
- Code that converts these raw data files into a format that's useable by your dataset subclass
- Dataset subclass that interfaces with your converted data

We'll cover the basics for the following use case:

- The data consists of several data sources (*e.g.* features, targets) that can be stored in `numpy.ndarray`-like objects
- Data sources have a fixed shape (*e.g.* vectors of size 100, images of width 32, weight 32 and with 3 channels)
- The data is split into various sets (*e.g.* training, validation, test)

5.1 Toy example

For this tutorial, we'll implement the venerable Iris dataset in Fuel. This dataset features 150 examples split into three classes (50 examples per class), and each example consists of four features.

For the purpose of demonstration, we'll split the dataset into a training set (100 examples), a validation set (20 examples) and a test set (30 examples). We'll pretend the test set doesn't have any label information available, as is often the case for machine learning competitions.

5.2 Download code

The Iris dataset is contained in a single file, iris.data, which we'll need to make available for users to download.

The preferred way of downloading dataset files in Fuel is the `fuel-download` script. Dataset implementations include a function for downloading their required files in the `fuel.downloaders` subpackage. In order to make that function accessible to `fuel-download`, they need to include it in the `all_downloaders` attribute of the `fuel.downloaders` subpackage.

The function accepts an `argparse.ArgumentParser` instance as input and should return a downloading function. Put the following piece of code inside the `fuel.downloaders.iris` module (you'll have to create it):

```python
from fuel.downloaders.base import default_downloader

def fill_subparser(subparser):
    subparser.set_defaults(
        urls=['https://archive.ics.uci.edu/ml/machine-learning-databases/'
              'iris/iris.data'],
        filenames=['iris.data'])
    return default_downloader
```

You should also register Iris as a downloadable dataset via the `all_downloaders` attribute. It's a tuple of pairs of name and subparser filler function. Here's an example of how the `fuel.downloaders` init file might look:

```python
from fuel.downloaders import binarized_mnist
from fuel.downloaders import iris

all_downloaders = (
    ('binarized_mnist', binarized_mnist.fill_subparser),
    ('iris', iris.fill_subparser))
```

A lot is going on in these few lines of code, so let's break it down.

In order to be more flexible, the `fuel-download` script uses subparsers. This lets each dataset define their own set of arguments. If you registered it properly, the function you just defined will get called and be given its own subparser to fill. Users will then be able to type the `fuel-download iris` command and `iris.data` will be downloaded.

When the `fuel-download iris` command is typed, the download script will call the function returned by `fill_subparser` and give it the `argparse.Namespace` instance containing all parsed command line arguments. That function is responsible for downloading the data.

We used the `default_downloader()` convenience function as our download function. It expects the parsed arguments to contain a list of URLs and a list of filenames, and downloads each URL, saving it under its corresponding filename. This is why we set the `urls` and `filenames` default arguments.

If your use case is more exotic, you can just as well define your own download function. Be aware of the following default arguments:

- `directory` : in which directory the files need to be saved
- `clear` : if `True`, your download function is expected to remove the downloaded files from `directory`.

5.3 Conversion code

In order to minimize the amount of code we have to write, we'll subclass `H5PYDataset`. This means we'll have to create an HDF5 file to store our data. For more information, see the *dedicated tutorial* on how to create an `H5PYDataset`-compatible HDF5 file.

Much like for downloading data files, the preferred way of converting data files in Fuel is through the `fuel-convert` script. Its implementation is very similar to `fuel-download`. The arguments to be aware of in the subparser are

- `directory` : in which directory the input files reside
- `output-directory` : where to save the converted dataset

The converter function should return a tuple containing path(s) to the converted dataset(s).

Put the following piece of code inside the `fuel.converters.iris` module (you'll have to create it):

```python
import os
import h5py
import numpy

from fuel.converters.base import fill_hdf5_file


def convert_iris(directory, output_directory, output_filename='iris.hdf5'):
    output_path = os.path.join(output_directory, output_filename)
    h5file = h5py.File(output_path, mode='w')
    classes = {'Iris-setosa': 0, 'Iris-versicolor': 1, 'Iris-virginica': 2}
    data = numpy.loadtxt(
        os.path.join(directory, 'iris.data'),
        converters={4: lambda x: classes[x]},
        delimiter=',')
    numpy.random.shuffle(data)
    features = data[:, :-1].astype('float32')
    targets = data[:, -1].astype('uint8')
    train_features = features[:100]
    train_targets = targets[:100]
    valid_features = features[100:120]
    valid_targets = targets[100:120]
    test_features = features[120:]
    data = (('train', 'features', train_features),
            ('train', 'targets', train_targets),
            ('valid', 'features', valid_features),
            ('valid', 'targets', valid_targets),
            ('test', 'features', test_features))
    fill_hdf5_file(h5file, data)
    h5file['features'].dims[0].label = 'batch'
    h5file['features'].dims[1].label = 'feature'
    h5file['targets'].dims[0].label = 'batch'
    h5file['targets'].dims[1].label = 'index'

    h5file.flush()
    h5file.close()

    return (output_path,)

def fill_subparser(subparser):
    return convert_iris
```

We used the convenience `fill_hdf5_file()` function to populate our HDF5 file and create the split array. This function expects a tuple of tuples, one per split/source pair, containing the split name, the source name, the data array and (optionally) a comment string.

We also used `H5PYDataset`'s ability to extract axis labels to add semantic information to the axes of our data sources. This allowed us to specify that target values are categorical (`'index`'). Note that you can use whatever label you want in Fuel, although certain frameworks using Fuel may have some hard-coded assumptions about which labels to use.

As for the download code, you should register Iris as a convertible dataset via the `all_converters` attribute of the `fuel.converters` subpackage. Here's an example of how the init file might look:

```
from fuel.converters import binarized_mnist
from fuel.converters import iris

all_converters = (
    ('binarized_mnist', binarized_mnist.fill_subparser),
    ('iris', iris.fill_subparser))
```

5.4 Dataset subclass

Let's now implement the `H5PYDataset` subclass that will interface with our newly-created HDF5 file.

One advantage of subclassing `H5PYDataset` is that the amount of code to write is very minimal:

```
from fuel.datasets import H5PYDataset
from fuel.utils import find_in_data_path


class Iris(H5PYDataset):
    filename = 'iris.hdf5'

    def __init__(self, which_sets=which_sets, **kwargs):
        kwargs.setdefault('load_in_memory', True)
        super(Iris, self).__init__(
            file_or_path=find_in_data_path(self.filename),
            which_sets=which_sets, **kwargs)
```

Our subclass is just a thin wrapper around the `H5PYDataset` class that defines the data path and switches the `load_in_memory` argument default to `True` (since this dataset easily fits in memory). Everything else is handled by the superclass.

5.5 Putting it together

We now have everything we need to start playing around with our new dataset implementation.

Try downloading and converting the data file:

```
cd $FUEL_DATA_PATH
fuel-download iris
fuel-convert iris
fuel-download iris --clear
cd -
```

You can now use the Iris dataset like you would use any other built-in dataset:

```
>>> from fuel.datasets.iris import Iris
>>> train_set = Iris(('train',))
>>> print(train_set.axis_labels['features'])
('batch', 'feature')
>>> print(train_set.axis_labels['targets'])
('batch', 'index')
>>> handle = train_set.open()
>>> data = train_set.get_data(handle, slice(0, 10))
>>> print((data[0].shape, data[1].shape))
((10, 4), (10, 1))
>>> train_set.close(handle)
```

5.6 Working with external packages

To distribute Fuel-compatible downloaders and converters independently from Fuel, you have to write your own modules or subpackages which will define `all_downloaders` and `all_converters`. Here is how the Iris downloader and converter might look like if you were to include them as part of the `my_fuel` package:

```
# my_fuel/downloaders/iris_downloader.py
from fuel.downloaders.base import default_downloader

def fill_subparser(subparser):
    subparser.set_defaults(
        urls=['https://archive.ics.uci.edu/ml/machine-learning-databases/'
              'iris/iris.data'],
        filenames=['iris.data'])
    return default_downloader
```

```
# my_fuel/downloaders/__init__.py
from my_fuel.downloaders import iris

all_downloaders = (('iris', iris.fill_subparser),)
```

```
# my_fuel/converters/iris.py
import os

import h5py
import numpy

from fuel.converters.base import fill_hdf5_file


def convert_iris(directory, output_directory, output_filename='iris.hdf5'):
    output_path = os.path.join(output_directory, output_filename)
    h5file = h5py.File(output_path, mode='w')
    classes = {'Iris-setosa': 0, 'Iris-versicolor': 1, 'Iris-virginica': 2}
    # ...

def fill_subparser(subparser):
    return convert_iris
```

```
# my_fuel/converters/__init__.py
from my_fuel.converters import iris

all_converters = (('iris', iris.fill_subparser),)
```

In order to use the downloaders and converters defined in `my_fuel`, users would then have to set the `extra_downloaders` and `extra_converters` configuration variables inside `~/.fuelrc` like so:

```
extra_downloaders: ['my_fuel.downloaders']
extra_converters: ['my_fuel.converters']
```

or set the `FUEL_EXTRA_DOWNLOADERS` and `FUEL_EXTRA_CONVERTERS` environment variables like so (this would override the `extra_downloaders` and `extra_converters` configuration variables):

```
$ export FUEL_EXTRA_DOWNLOADERS=my_fuel.downloaders
$ export FUEL_EXTRA_CONVERTERS=my_fuel.converters
```

CHAPTER 6

Extending Fuel

In this section, we'll cover how to extend three important components of Fuel:

- Dataset classes
- Iteration schemes
- Transformers

Warning: The code presented in this section is for illustration purposes only and is not intended to be used outside of this tutorial.

6.1 New dataset classes

In summary

- Subclass from `Dataset`.
  - Implement the `get_data()` method.
  - If your dataset interacts with stateful objects (e.g. files on disk), override the `open()` and `close()` methods.
- Subclass from `IndexableDataset` if your data fits in memory.
  - `IndexableDataset` constructor accepts an `indexables` argument, which is expected to be a `dict` mapping from source names to their corresponding data.

New dataset classes are implemented by subclassing `Dataset` and implementing its `get_data()` method. If your dataset interacts with stateful objects (e.g. files on disk), then you should also override the `open()` and `close()` methods.

If your data fits in memory, you can save yourself some time by inheriting from `IndexableDataset`. In that case, all you need to do is load the data as a `dict` mapping source names to their corresponding data and pass it to the superclass as the `indexables` argument.

For instance, here's how you would implement a specialized class to interface with `.npy` files.

```
>>> from collections import OrderedDict
>>> import numpy
>>> from six import iteritems
>>> from fuel.datasets import IndexableDataset
```

```
>>>
>>> class NPYDataset(IndexableDataset):
...     def __init__(self, source_paths, **kwargs):
...         indexables = OrderedDict(
...             [(source, numpy.load(path)) for
...              source, path in iteritems(source_paths)])
...         super(NPYDataset, self).__init__(indexables, **kwargs)
```

Here's this class in action:

```
>>> numpy.save('npy_dataset_features.npy',
...            numpy.arange(40).reshape((10, 4)))
>>> numpy.save('npy_dataset_targets.npy',
...            numpy.arange(10).reshape((10, 1)))
>>> dataset = NPYDataset(OrderedDict([('features', 'npy_dataset_features.npy'),
...                                   ('targets', 'npy_dataset_targets.npy')]))
>>> state = dataset.open()
>>> print(dataset.get_data(state=state, request=[0, 1, 2, 3]))
(array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]]), array([[0],
       [1],
       [2],
       [3]]))
>>> dataset.close(state)
```

6.2 New iteration schemes

In summary

- Subclass from `IterationScheme`.
  - Implement the `get_request_iterator()` method, which should return an iterator object.
- Subclass from `IndexScheme` for schemes requesting examples.
  - `IndexScheme` constructor accepts an `examples` argument, which can either be an integer or a list of indices.
- Subclass from `BatchScheme` for schemes requesting batches.
  - `BatchScheme` constructor accepts an `examples` argument, which can either be an integer or a list of indices, and a `batch_size` argument.

New iteration schemes are implemented by subclassing `IterationScheme` and implementing a `get_request_iterator()` method, which should return an iterator that returns lists of indices.

Two subclasses of `IterationScheme` typically serve as a basis for other iteration schemes: `IndexScheme` (for schemes requesting examples) and `BatchScheme` (for schemes requesting batches). Both subclasses are instantiated by providing a list of indices or the number of examples to visit (meaning that examples 0 through `examples - 1` would be visited), and `BatchScheme` accepts an additional `batch_size` argument.

Here's how you would implement an iteration scheme that iterates over even examples:

```
>>> from fuel.schemes import IndexScheme, BatchScheme
>>> # `iter_` : A picklable version of `iter`
>>> from picklable_itertools import iter_, imap
>>> # Partition all elements of a sequence into tuples of length at most n
>>> from picklable_itertools.extras import partition_all
```

```
>>> class ExampleEvenScheme(IndexScheme):
...     def get_request_iterator(self):
...         indices = list(self.indices)[::2]
...         return iter_(indices)
>>> class BatchEvenScheme(BatchScheme):
...     def get_request_iterator(self):
...         indices = list(self.indices)[::2]
...         return imap(list, partition_all(self.batch_size, indices))
```

Note: The `partition_all` function splits a sequence into chunks of size `n` (`self.batch_size`, in our case), with the last batch possibly being shorter if the length of the sequence is not a multiple of `n`. It produces an iterator which returns these batches as tuples.

The `imap` function applies a function to all elements of a sequence. It produces an iterator which returns the result of the function being applied to elements of the sequence. In our case, this has the effect of casting tuples into lists.

The reason why we go through all this trouble is to produce an iterator, which is what `get_request_iterator()` is supposed to return.

Here are the two iteration scheme classes in action:

```
>>> print(list(ExampleEvenScheme(10).get_request_iterator()))
[0, 2, 4, 6, 8]
>>> print(list(BatchEvenScheme(10, 2).get_request_iterator()))
[[0, 2], [4, 6], [8]]
```

6.3 New transformers

In summary

- Subclass from `Transformer`.
  - Implement `transform_example()` if your transformer works on single examples and/or `transform_batch()` if it works on batches.
- Subclass from `AgnosticTransformer` if the example and batch implementations are the same.
  - Implement the `transform_any()` method.
- Subclass from `SourcewiseTransformer` if your transformation is applied sourcewise.
  - `SourcewiseTransformer` constructor takes an additional `which_sources` keyword argument.
  - If `which_sources` is `None`, then the transformation is applied to all sources.
  - Implement the `transform_source_example()` and/or `transform_source_batch()` methods.
- Subclass from `AgnosticSourcewiseTransformer` if your transformation is applied sourcewise and its example and batch implementations are the same.
  - Implement the `transform_any_source()` method.

- Consider using the `Mapping` transformer (the swiss-army knife of transformers) for one-off transformations instead of implementing a subclass.
  - Its constructor accepts a `mapping` argument, which is expected to be a function taking a tuple of sources as input and returning the transformed sources.

An important thing to know about data streams is that they distinguish between two types of outputs: single examples, and batches of examples. Depending on your choice of iteration scheme, a data stream's `produces_examples` property will either be `True` (it produces examples) or `False` (it produces batches).

Transformers are aware of this, and as such implement two distinct methods: `transform_example()` and `transform_batch()`. A new transformer is typically implemented by subclassing Transformer and implementing one or both of these methods.

As an example, here's how you would double the value of some `'features'` data source.

```
>>> from fuel.transformers import Transformer
>>>
>>> class FeaturesDoubler(Transformer):
...     def __init__(self, data_stream, **kwargs):
...         super(FeaturesDoubler, self).__init__(
...             data_stream=data_stream,
...             produces_examples=data_stream.produces_examples,
...             **kwargs)
...
...     def transform_example(self, example):
...         if 'features' in self.sources:
...             example = list(example)
...             index = self.sources.index('features')
...             example[index] *= 2
...             example = tuple(example)
...         return example
...
...     def transform_batch(self, batch):
...         if 'features' in self.sources:
...             batch = list(batch)
...             index = self.sources.index('features')
...             batch[index] *= 2
...             batch = tuple(batch)
...         return batch
```

Since we wish to support both batches and examples, we'll declare our output type to be the same as our data stream's output type.

If you were to build a transformer that only works on batches, you would pass `produces_examples=False` and implement only `transform_batch()`. If anyone tried to use your transformer on an example data stream, an error would automatically be raised.

Note: When applicable, it is good practice that a new transformer's constructor calls its superclass constructor by passing the data stream it receives and declaring whether it produce examples or batches.

Let's test our doubler on some dummy dataset.

```
>>> from fuel.schemes import SequentialExampleScheme, SequentialScheme
>>> from fuel.streams import DataStream
>>>
>>> dataset = IndexableDataset(
...     indexables=OrderedDict([
...         ('features', numpy.array([1, 2, 3, 4])),
...         ('targets', numpy.array([-1, 1, -1, 1]))]))
>>> example_scheme = SequentialExampleScheme(examples=dataset.num_examples)
>>> example_stream = FeaturesDoubler(
...     data_stream=DataStream(
...         dataset=dataset, iteration_scheme=example_scheme))
>>> batch_scheme = SequentialScheme(
...     examples=dataset.num_examples, batch_size=2)
>>> batch_stream = FeaturesDoubler(
...     data_stream=DataStream(
...         dataset=dataset, iteration_scheme=batch_scheme))
>>> print([example for example in example_stream.get_epoch_iterator()])
[(2, -1), (4, 1), (6, -1), (8, 1)]
>>> print([batch for batch in batch_stream.get_epoch_iterator()])
[(array([2, 4]), array([-1,  1])), (array([6, 8]), array([-1,  1]))]
```

You may have noticed that in this example the `transform_example()` and `transform_batch()` implementations are the same. In such cases, you can subclass from `AgnosticTransformer` instead. It requires that you implement a `transform_any()` method, which will be called by both `transform_example()` and `transform_batch()`.

```
>>> from fuel.transformers import AgnosticTransformer
>>>
>>> class FeaturesDoubler(AgnosticTransformer):
...     def __init__(self, data_stream, **kwargs):
...         super(FeaturesDoubler, self).__init__(
...             data_stream=data_stream,
...             produces_examples=data_stream.produces_examples,
...             **kwargs)
...
...     def transform_any(self, data):
...         if 'features' in self.sources:
...             data = list(data)
...             index = self.sources.index('features')
...             data[index] *= 2
...             data = tuple(data)
...         return data
```

So far so good, but our transformer applies the same transformation to every source in the dataset. What if we want to be more general and be able to select which sources we want to process with our transformer?

Transformers which are applied sourcewise like our doubler should usually subclass from `SourcewiseTransformer`. Their constructor takes an additional `which_sources` keyword argument specifying which sources to apply the transformer to. It's expected to be a tuple of source names. If `which_sources` is `None`, then the transformer is applied to all sources. Subclasses of `SourcewiseTransformer` should implement a `transform_source_example()` method and/or a `transform_source_batch()` method, which apply on an individual source.

There also exists an `AgnosticSourcewiseTransformer` class for cases where the example and batch implementations of a sourcewise transformer are the same. This class requires a `transform_any_source()` method to be implemented.

```
>>> from fuel.transformers import AgnosticSourcewiseTransformer
>>>
>>> class Doubler(AgnosticSourcewiseTransformer):
...     def __init__(self, data_stream, **kwargs):
...         super(Doubler, self).__init__(
...             data_stream=data_stream,
...             produces_examples=data_stream.produces_examples,
...                **kwargs)
...
...     def transform_any_source(self, source, _):
...         return 2 * source
```

Let's try this implementation on our dummy dataset.

```
>>> target_stream = Doubler(
...     data_stream=DataStream(
...         dataset=dataset,
...         iteration_scheme=batch_scheme),
...     which_sources=('targets',))
>>> all_stream = Doubler(
...     data_stream=DataStream(
...         dataset=dataset,
...         iteration_scheme=batch_scheme),
...     which_sources=None)
>>> print([batch for batch in target_stream.get_epoch_iterator()])
[(array([1, 2]), array([-2,  2])), (array([3, 4]), array([-2,  2]))]
>>> print([batch for batch in all_stream.get_epoch_iterator()])
[(array([2, 4]), array([-2,  2])), (array([6, 8]), array([-2,  2]))]
```

Finally, what if we not only want to select at runtime which sources the transformation applies to, but also the transformer itself? This is what the `Mapping` transformer solves. In addition to a data stream, its constructor accepts a `mapping` argument, which is expected to be a function. This function which will be applied to data coming from the stream.

Here's how you would implement the feature doubler using `Mapping`.

```
>>> from fuel.transformers import Mapping
>>>
>>> features_index = dataset.sources.index('features')
>>> def double(data):
...     data = list(data)
...     data[features_index] *= 2
...     return tuple(data)
>>> mapping_stream = Mapping(
...     data_stream=DataStream(
...         dataset=dataset, iteration_scheme=batch_scheme),
...     mapping=double)
>>> print([batch for batch in mapping_stream.get_epoch_iterator()])
[(array([2, 4]), array([-1,  1])), (array([6, 8]), array([-1,  1]))]
```

CHAPTER 7

API Reference

Warning: This API reference is currently nothing but a dump of docstrings, ordered alphabetically.

7.1 Converters

7.1.1 Base classes

`fuel.converters.base.`**`check_exists`**`(`*required_files*`)`

Decorator that checks if required files exist before running.

Parameters **`required_files`** (*`list of str`*) – A list of strings indicating the filenames of regular files (not directories) that should be found in the input directory (which is the first argument to the wrapped function).

Returns **wrapper** – A function that takes a function and returns a wrapped function. The function returned by *wrapper* will include input file existence verification.

Return type function

Notes

Assumes that the directory in which to find the input files is provided as the first argument, with the argument name *directory*.

`fuel.converters.base.`**`fill_hdf5_file`**`(`*h5file*, *data*`)`

Fills an HDF5 file in a H5PYDataset-compatible manner.

Parameters

- **`h5file`** (`h5py.File`) – File handle for an HDF5 file.
- **`data`** (*`tuple of tuple`*) – One element per split/source pair. Each element consists of a tuple of (split_name, source_name, data_array, comment), where
  - ‘split_name’ is a string identifier for the split name
  - ‘source_name’ is a string identifier for the source name
  - ‘data_array’ is a `numpy.ndarray` containing the data for this split/source pair
  - ‘comment’ is a comment string for the split/source pair

The ‘comment’ element can optionally be omitted.

`fuel.converters.base.`**`progress_bar`**`(`**args*, ***kwds*`)`

Manages a progress bar for a conversion.

Parameters

- **`name`** (*`str`*) – Name of the file being converted.
- **`maxval`** (*`int`*) – Total number of steps for the conversion.

7.1.2 Adult

`fuel.converters.adult.`**`convert_adult`**`(`*directory*, *output_directory*, *output_filename='adult.hdf5'*`)`

Convert the Adult dataset to HDF5.

Converts the Adult dataset to an HDF5 dataset compatible with `fuel.datasets.Adult`. The converted dataset is saved as ‘adult.hdf5’. This method assumes the existence of the file *adult.data* and *adult.test*.

Parameters

- **`directory`** (*`str`*) – Directory in which input files reside.
- **`output_directory`** (*`str`*) – Directory in which to save the converted dataset.
- **`output_filename`** (*`str, optional`*) – Name of the saved dataset. Defaults to *adult.hdf5*.

Returns **output_paths** – Single-element tuple containing the path to the converted dataset.

Return type tuple of str

`fuel.converters.adult.`**`convert_to_one_hot`**`(`*y*`)`

converts y into one hot reprsentation.

Parameters **`y`** (*`list`*) – A list containing continous integer values.

Returns **one_hot** – A numpy.ndarray object, which is one-hot representation of y.

Return type numpy.ndarray

`fuel.converters.adult.`**`fill_subparser`**`(`*subparser*`)`

7.1.3 CalTech 101 Silhouettes

`fuel.converters.caltech101_silhouettes.`**`convert_silhouettes`**`(`*size*, *directory*, *output_directory*, *output_file=None*`)`

Convert the CalTech 101 Silhouettes Datasets.

Parameters

- **`size`** (*`{16, 28}`*) – Convert either the 16x16 or 28x28 sized version of the dataset.
- **`directory`** (*`str`*) – Directory in which the required input files reside.
- **`output_file`** (*`str`*) – Where to save the converted dataset.

`fuel.converters.caltech101_silhouettes.`**`fill_subparser`**`(`*subparser*`)`

Sets up a subparser to convert CalTech101 Silhouettes Database files.

Parameters **`subparser`** (`argparse.ArgumentParser`) – Subparser handling the *caltech101_silhouettes* command.

7.1.4 Binarized MNIST

`fuel.converters.binarized_mnist.`**`convert_binarized_mnist`**(*directory*, **args*, ***kwargs*)

Converts the binarized MNIST dataset to HDF5.

Converts the binarized MNIST dataset used in R. Salakhutdinov's DBN paper [DBN] to an HDF5 dataset compatible with `fuel.datasets.BinarizedMNIST`. The converted dataset is saved as 'binarized_mnist.hdf5'.

This method assumes the existence of the files *binarized_mnist_{train,valid,test}.amat*, which are accessible through Hugo Larochelle's website [HUGO].

Parameters

- **`directory`** (`str`) – Directory in which input files reside.
- **`output_directory`** (`str`) – Directory in which to save the converted dataset.
- **`output_filename`** (`str, optional`) – Name of the saved dataset. Defaults to 'binarized_mnist.hdf5'.

Returns **output_paths** – Single-element tuple containing the path to the converted dataset.

Return type tuple of str

`fuel.converters.binarized_mnist.`**`fill_subparser`**(*subparser*)

Sets up a subparser to convert the binarized MNIST dataset files.

Parameters **`subparser`** (`argparse.ArgumentParser`) – Subparser handling the *binarized_mnist* command.

7.1.5 CIFAR100

`fuel.converters.cifar100.`**`convert_cifar100`**(*directory*, **args*, ***kwargs*)

Converts the CIFAR-100 dataset to HDF5.

Converts the CIFAR-100 dataset to an HDF5 dataset compatible with `fuel.datasets.CIFAR100`. The converted dataset is saved as 'cifar100.hdf5'.

This method assumes the existence of the following file: *cifar-100-python.tar.gz*

Parameters

- **`directory`** (`str`) – Directory in which the required input files reside.
- **`output_directory`** (`str`) – Directory in which to save the converted dataset.
- **`output_filename`** (`str, optional`) – Name of the saved dataset. Defaults to 'cifar100.hdf5'.

Returns **output_paths** – Single-element tuple containing the path to the converted dataset.

Return type tuple of str

`fuel.converters.cifar100.`**`fill_subparser`**(*subparser*)

Sets up a subparser to convert the CIFAR100 dataset files.

Parameters **`subparser`** (`argparse.ArgumentParser`) – Subparser handling the *cifar100* command.

7.1.6 CIFAR10

`fuel.converters.cifar10.`**`convert_cifar10`**(*directory*, **args*, ***kwargs*)

Converts the CIFAR-10 dataset to HDF5.

Converts the CIFAR-10 dataset to an HDF5 dataset compatible with `fuel.datasets.CIFAR10`. The converted dataset is saved as 'cifar10.hdf5'.

It assumes the existence of the following file:

- *cifar-10-python.tar.gz*

Parameters

- **`directory`** (*`str`*) – Directory in which input files reside.
- **`output_directory`** (*`str`*) – Directory in which to save the converted dataset.
- **`output_filename`** (*`str, optional`*) – Name of the saved dataset. Defaults to 'cifar10.hdf5'.

Returns **output_paths** – Single-element tuple containing the path to the converted dataset.

Return type tuple of str

`fuel.converters.cifar10.`**`fill_subparser`**(*subparser*)

Sets up a subparser to convert the CIFAR10 dataset files.

Parameters **`subparser`** (`argparse.ArgumentParser`) – Subparser handling the *cifar10* command.

7.1.7 IRIS

`fuel.converters.iris.`**`convert_iris`**(*directory*, *output_directory*, *output_filename='iris.hdf5'*)

Convert the Iris dataset to HDF5.

Converts the Iris dataset to an HDF5 dataset compatible with `fuel.datasets.Iris`. The converted dataset is saved as 'iris.hdf5'. This method assumes the existence of the file *iris.data*.

Parameters

- **`directory`** (*`str`*) – Directory in which input files reside.
- **`output_directory`** (*`str`*) – Directory in which to save the converted dataset.
- **`output_filename`** (*`str, optional`*) – Name of the saved dataset. Defaults to *None*, in which case a name based on *dtype* will be used.

Returns **output_paths** – Single-element tuple containing the path to the converted dataset.

Return type tuple of str

`fuel.converters.iris.`**`fill_subparser`**(*subparser*)

Sets up a subparser to convert the Iris dataset file.

Parameters **`subparser`** (`argparse.ArgumentParser`) – Subparser handling the *iris* command.

7.1.8 MNIST

`fuel.converters.mnist.`**`convert_mnist`**(*directory*, **args*, ***kwargs*)

Converts the MNIST dataset to HDF5.

Converts the MNIST dataset to an HDF5 dataset compatible with `fuel.datasets.MNIST`. The converted dataset is saved as 'mnist.hdf5'.

This method assumes the existence of the following files: *train-images-idx3-ubyte.gz*, *train-labels-idx1-ubyte.gz* *t10k-images-idx3-ubyte.gz*, *t10k-labels-idx1-ubyte.gz*

It assumes the existence of the following files:

- *train-images-idx3-ubyte.gz*
- *train-labels-idx1-ubyte.gz*
- *t10k-images-idx3-ubyte.gz*
- *t10k-labels-idx1-ubyte.gz*

Parameters

- **`directory`** (*`str`*) – Directory in which input files reside.
- **`output_directory`** (*`str`*) – Directory in which to save the converted dataset.
- **`output_filename`** (*`str, optional`*) – Name of the saved dataset. Defaults to *None*, in which case a name based on *dtype* will be used.
- **`dtype`** (*`str, optional`*) – Either 'float32', 'float64', or 'bool'. Defaults to *None*, in which case images will be returned in their original unsigned byte format.

Returns output_paths – Single-element tuple containing the path to the converted dataset.

Return type tuple of str

`fuel.converters.mnist.`**`fill_subparser`**(*subparser*)

Sets up a subparser to convert the MNIST dataset files.

Parameters subparser (`argparse.ArgumentParser`) – Subparser handling the *mnist* command.

`fuel.converters.mnist.`**`read_mnist_images`**(*filename*, *dtype=None*)

Read MNIST images from the original ubyte file format.

Parameters

- **`filename`** (*`str`*) – Filename/path from which to read images.
- **`dtype`** (*`'float32', 'float64', or 'bool'`*) – If unspecified, images will be returned in their original unsigned byte format.

Returns images – An image array, with individual examples indexed along the first axis and the image dimensions along the second and third axis.

Return type `ndarray`, shape (n_images, 1, n_rows, n_cols)

Notes

If the dtype provided was Boolean, the resulting array will be Boolean with *True* if the corresponding pixel had a value greater than or equal to 128, *False* otherwise.

If the dtype provided was a float dtype, the values will be mapped to the unit interval [0, 1], with pixel values that were 255 in the original unsigned byte representation equal to 1.0.

`fuel.converters.mnist.`**`read_mnist_labels`**(*filename*)

Read MNIST labels from the original ubyte file format.

Parameters **`filename`** (*`str`*) – Filename/path from which to read labels.

Returns **labels** – A one-dimensional unsigned byte array containing the labels as integers.

Return type `ndarray`, shape (nlabels, 1)

7.1.9 SVHN

`fuel.converters.svhn.`**`convert_svhn`**(*which_format*, *directory*, *output_directory*, *output_filename=None*)

Converts the SVHN dataset to HDF5.

Converts the SVHN dataset [SVHN] to an HDF5 dataset compatible with `fuel.datasets.SVHN`. The converted dataset is saved as ‘svhn_format_1.hdf5’ or ‘svhn_format_2.hdf5’, depending on the *which_format* argument.

Parameters

- **`which_format`** (*`int`*) – Either 1 or 2. Determines which format (format 1: full numbers or format 2: cropped digits) to convert.
- **`directory`** (*`str`*) – Directory in which input files reside.
- **`output_directory`** (*`str`*) – Directory in which to save the converted dataset.
- **`output_filename`** (*`str, optional`*) – Name of the saved dataset. Defaults to ‘svhn_format_1.hdf5’ or ‘svhn_format_2.hdf5’, depending on *which_format*.

Returns **output_paths** – Single-element tuple containing the path to the converted dataset.

Return type tuple of str

`fuel.converters.svhn.`**`convert_svhn_format_1`**(*directory*, **args*, ***kwargs*)

Converts the SVHN dataset (format 1) to HDF5.

This method assumes the existence of the files *{train,test,extra}.tar.gz*, which are accessible through the official website [SVHNSITE].

Parameters

- **`directory`** (*`str`*) – Directory in which input files reside.
- **`output_directory`** (*`str`*) – Directory in which to save the converted dataset.
- **`output_filename`** (*`str, optional`*) – Name of the saved dataset. Defaults to ‘svhn_format_1.hdf5’.

Returns **output_paths** – Single-element tuple containing the path to the converted dataset.

Return type tuple of str

`fuel.converters.svhn.`**`convert_svhn_format_2`**(*directory*, **args*, ***kwargs*)

Converts the SVHN dataset (format 2) to HDF5.

This method assumes the existence of the files *{train,test,extra}_32x32.mat*, which are accessible through the official website [SVHNSITE].

Parameters

- **directory** (*str*) – Directory in which input files reside.
- **output_directory** (*str*) – Directory in which to save the converted dataset.
- **output_filename** (*str, optional*) – Name of the saved dataset. Defaults to 'svhn_format_2.hdf5'.

Returns **output_paths** – Single-element tuple containing the path to the converted dataset.

Return type tuple of str

`fuel.converters.svhn.`**`fill_subparser`**(*subparser*)
Sets up a subparser to convert the SVHN dataset files.

Parameters **subparser** (`argparse.ArgumentParser`) – Subparser handling the *svhn* command.

7.2 Data streams

class `fuel.streams.`**`AbstractDataStream`**(*iteration_scheme=None*, *axis_labels=None*)
Bases: `object`

A stream of data separated into epochs.

A data stream is an iterable stream of examples/minibatches. It shares similarities with Python file handles return by the `open` method. Data streams can be closed using the *`close()`* method and reset using *`reset()`* (similar to `f.seek(0)`).

Parameters

- **iteration_scheme** (*IterationScheme*, optional) – The iteration scheme to use when retrieving data. Note that not all datasets support the same iteration schemes, some datasets require one, and others don't support any. In case when the data stream wraps another data stream, the choice of supported iteration schemes is typically even more limited. Be sure to read the documentation of the dataset or data stream in question.
- **axis_labels** (*dict, optional*) – Maps source names to tuples of strings describing axis semantics, one per axis. Defaults to *None*, i.e. no information is available.

iteration_scheme
IterationScheme

The iteration scheme used to retrieve data. Can be `None` when not used.

sources
tuple of strings

The names of the data sources returned by this data stream, as given by the dataset.

produces_examples
bool

Whether this data stream produces examples (as opposed to batches of examples).

close()
Gracefully close the data stream, e.g. releasing file handles.

get_data(*request=None*)
Request data from the dataset or the wrapped stream.

Parameters **request** (*object*) – A request fetched from the *request_iterator*.

get_epoch_iterator(*as_dict=False*)

iterate_epochs(*as_dict=False*)
Allow iteration through all epochs.

Notes

This method uses the *get_epoch_iterator()* method to retrieve the `DataIterator` for each epoch. The default implementation of this method resets the state of the data stream so that the new epoch can read the data from the beginning. However, this behavior only works as long as the `epochs` property is iterated over using e.g. `for epoch in stream.epochs`. If you create the data iterators in advance (e.g. using `for i, epoch in zip(range(10), stream.epochs` in Python 2) you must call the *reset()* method yourself.

next_epoch()
Switch the data stream to the next epoch.

produces_examples

reset()
Reset the data stream.

class fuel.streams.**DataStream**(*dataset*, ***kwargs*)
Bases: *fuel.streams.AbstractDataStream*

A stream of data from a dataset.

Parameters **dataset** (instance of `Dataset`) – The dataset from which the data is fetched.

close()

classmethod default_stream(*dataset*, ***kwargs*)

get_data(*request=None*)
Get data from the dataset.

get_epoch_iterator(***kwargs*)
Get an epoch iterator for the data stream.

next_epoch()

reset()

sources

class fuel.streams.**ServerDataStream**(*sources*, *produces_examples*, *host='localhost'*, *port=5557*, *hwm=10*, *axis_labels=None*)
Bases: *fuel.streams.AbstractDataStream*

A data stream that receives batches from a Fuel server.

Parameters

- **sources** (*tuple of strings*) – The names of the data sources returned by this data stream.
- **produces_examples** (*bool*) – Whether this data stream produces examples (as opposed to batches of examples).
- **host** (*str, optional*) – The host to connect to. Defaults to `localhost`.
- **port** (*int, optional*) – The port to connect on. Defaults to 5557.
- **hwm** (*int, optional*) – The ZeroMQ high-water mark (HWM) on the receiving socket. Increasing this increases the buffer, which can be useful if your data preprocessing times are very random. However, it will increase memory usage. There is no easy way to tell how

many batches will actually be queued with a particular HWM. Defaults to 10. Be sure to set the corresponding HWM on the server's end as well.

- **axis_labels** (`dict, optional`) – Maps source names to tuples of strings describing axis semantics, one per axis. Defaults to *None*, i.e. no information is available.

close()

connect()

get_data(*request=None*)

get_epoch_iterator(***kwargs*)

next_epoch()

reset()

7.3 Datasets

7.3.1 Base classes

class `fuel.datasets.base.`**Dataset**(*sources=None*, *axis_labels=None*)

Bases: `object`

A dataset.

Dataset classes implement the interface to a particular dataset. The interface consists of a number of routines to manipulate so called "state" objects, e.g. open, reset and close them.

Parameters

- **sources** (`tuple of strings, optional`) – The data sources to load and return by `get_data()`. By default all data sources are returned.
- **axis_labels** (`dict, optional`) – Maps source names to tuples of strings describing axis semantics, one per axis. Defaults to *None*, i.e. no information is available.

sources

tuple of strings

The sources this dataset will provide when queried for data e.g. `('features',)` when querying only the data from MNIST.

provides_sources

tuple of strings

The sources this dataset *is able to* provide e.g. `('features', 'targets')` for MNIST (regardless of which data the data stream actually requests). Any implementation of a dataset should set this attribute on the class (or at least before calling `super`).

example_iteration_scheme

`IterationScheme` or `None`

The iteration scheme the class uses in order to produce a stream of examples.

default_transformers

It is expected to be a tuple with one element per

transformer in the pipeline. Each element is a tuple with three elements:

- the Transformer subclass to apply,

- a list of arguments to pass to the subclass constructor, and
- a dict of keyword arguments to pass to the subclass constructor.

Notes

Datasets should only implement the interface; they are not expected to perform the iteration over the actual data. As such, they are stateless, and can be shared by different parts of the library simultaneously.

apply_default_transformers(*stream*)
Applies default transformers to a stream.

Parameters **stream** (`AbstractDataStream`) – A data stream.

close(*state*)
Cleanly close the dataset e.g. close file handles.

Parameters **state** (`object`) – The current state.

default_transformers = ()

example_iteration_scheme

filter_sources(*data*)
Filter the requested sources from those provided by the dataset.

A dataset can be asked to provide only a subset of the sources it can provide (e.g. asking MNIST only for the features, not for the labels). A dataset can choose to use this information to e.g. only load the requested sources into memory. However, in case the performance gain of doing so would be negligible, the dataset can load all the data sources and then use this method to return only those requested.

Parameters **data** (`tuple of objects`) – The data from all the sources i.e. should be of the same length as `provides_sources`.

Returns A tuple of data matching `sources`.

Return type tuple

Examples

```
>>> import numpy
>>> class Random(Dataset):
...     provides_sources = ('features', 'targets')
...     def get_data(self, state=None, request=None):
...         data = (numpy.random.rand(10), numpy.random.randn(3))
...         return self.filter_sources(data)
>>> Random(sources=('targets',)).get_data()
(array([-1.82436737,  0.08265948,  0.63206168]),)
```

get_data(*state=None*, *request=None*)
Request data from the dataset.

Todo

A way for the dataset to communicate which kind of requests it accepts, and a way to communicate what kind of request is being sent when supporting multiple.

Parameters

- **state** (*object, optional*) – The state as returned by the *open()* method. The dataset can use this to e.g. interact with files when needed.
- **request** (*object, optional*) – If supported, the request for a particular part of the data e.g. the number of examples to return, or the indices of a particular minibatch of examples.

Returns A tuple of data matching the order of *sources*.

Return type tuple

get_example_stream()

next_epoch(*state*)

Switches the dataset state to the next epoch.

The default implementation for this method is to reset the state.

Parameters **state** (*object*) – The current state.

Returns **state** – The state for the next epoch.

Return type object

open()

Return the state if the dataset requires one.

Datasets which e.g. read files from disks require open file handlers, and this sort of stateful information should be handled by the data stream.

Returns **state** – An object representing the state of a dataset.

Return type object

provides_sources = None

reset(*state*)

Resets the state.

Parameters **state** (*object*) – The current state.

Returns **state** – A reset state.

Return type object

Notes

The default implementation closes the state and opens a new one. A more efficient implementation (e.g. using `file.seek(0)` instead of closing and re-opening the file) can override the default one in derived classes.

sources

class fuel.datasets.base.**IndexableDataset**(*indexables*, *start=None*, *stop=None*, ***kwargs*)

Bases: *fuel.datasets.base.Dataset*

Creates a dataset from a set of indexable containers.

Parameters **indexables** (OrderedDict or indexable) – The indexable(s) to provide interface to. This means it must support the syntax `'indexable[0]`. If an OrderedDict is given, its values should be indexables providing data, and its keys strings that are used as source names. If a single indexable is given, it will be given the source `data`.

`indexables`
list

A list of indexable objects.

Notes

If the indexable data is very large, you might want to consider using the `do_not_pickle_attributes()` decorator to make sure the data doesn't get pickled with the dataset, but gets reloaded/recreated instead.

This dataset also uses the source names to create properties that provide easy access to the data.

`get_data`(*state=None*, *request=None*)

`num_examples`

class `fuel.datasets.base.`**`IterableDataset`**(*iterables*, ***kwargs*)
Bases: `fuel.datasets.base.Dataset`

Creates a dataset from a set of iterables.

Parameters **`iterables`** (`OrderedDict` or iterable) – The iterable(s) to provide interface to. The iterables' *__iter__* method should return a new iterator over the iterable. If an `OrderedDict` is given, its values should be iterables providing data, and its keys strings that are used as source names. If a single iterable is given, it will be given the source `data`.

`iterables`
list

A list of `Iterable` objects.

Notes

Internally, this method uses picklable iterools's `_iter` function, providing picklable alternatives to some iterators such as `range()`, `tuple()`, and even `file`. However, if the iterable returns a different kind of iterator that is not picklable, you might want to consider using the `do_not_pickle_attributes()` decorator.

To iterate over a container in batches, combine this dataset with the `Batch` data stream.

`example_iteration_scheme` = **None**

`get_data`(*state=None*, *request=None*)

`num_examples`

`open`()

7.3.2 Adult

class `fuel.datasets.adult.`**`Adult`**(*which_sets*, ***kwargs*)
Bases: `fuel.datasets.hdf5.H5PYDataset`

`filename` = **'adult.hdf5'**

7.3.3 One Billion Word

class `fuel.datasets.billion.`**`OneBillionWord`**(*which_set*, *which_partitions*, *dictionary*, ***kwargs*)

Bases: `fuel.datasets.text.TextFile`

Google's One Billion Word benchmark.

This monolingual corpus contains 829,250,940 tokens (including sentence boundary markers). The data is split into 100 partitions, one of which is the held-out set. This held-out set is further divided into 50 partitions. More information about the dataset can be found in [CMSG14].

Parameters

- **which_set** (*'training' or 'heldout'*) – Which dataset to load.
- **which_partitions** (*list of ints*) – For the training set, valid values must lie in [1, 99]. For the heldout set they must be in [0, 49].
- **vocabulary** (*dict*) – A dictionary mapping tokens to integers. This dictionary is expected to contain the tokens `<S>`, `</S>` and `<UNK>`, representing "start of sentence", "end of sentence", and "out-of-vocabulary" (OoV). The latter will be used whenever a token cannot be found in the vocabulary.
- **preprocess** (*function, optional*) – A function that takes a string (a sentence including new line) as an input and returns a modified string. A useful function to pass could be `str.lower`.

:param See `TextFile` for remaining keyword arguments.:

7.3.4 CalTech 101 Silhouettes

class `fuel.datasets.caltech101_silhouettes.`**`CalTech101Silhouettes`**(*which_sets*, *size=28*, *load_in_memory=True*, ***kwargs*)

Bases: `fuel.datasets.hdf5.H5PYDataset`

CalTech 101 Silhouettes dataset.

This dataset provides the *split1* train/validation/test split of the CalTech101 Silhouette dataset prepared by Benjamin M. Marlin [MARLIN].

This class provides both the 16x16 and the 28x28 pixel sized version. The 16x16 version contains 4082 examples in the training set, 2257 examples in the validation set and 2302 examples in the test set. The 28x28 version contains 4100, 2264 and 2307 examples in the train, valid and test set.

Parameters

- **which_sets** (*tuple of str*) – Which split to load. Valid values are 'train', 'valid' and 'test'.
- **size** (*{16, 28}*) – Either 16 or 28 to select the 16x16 or 28x28 pixels version of the dataset (default: 28).

data_path

7.3.5 Binarized MNIST

class `fuel.datasets.binarized_mnist.`**`BinarizedMNIST`**(*which_sets*, *load_in_memory=True*, ***kwargs*)

Bases: `fuel.datasets.hdf5.H5PYDataset`

Binarized, unlabeled MNIST dataset.

MNIST (Mixed National Institute of Standards and Technology) [LBBH] is a database of handwritten digits. It is one of the most famous datasets in machine learning and consists of 60,000 training images and 10,000 testing images. The images are grayscale and 28 x 28 pixels large.

This particular version of the dataset is the one used in R. Salakhutdinov's DBN paper [DBN] as well as the VAE and NADE papers, and is accessible through Hugo Larochelle's public website [HUGO].

The training set has further been split into a training and a validation set. All examples were binarized by sampling from a binomial distribution defined by the pixel values.

Parameters **`which_sets`** (*`tuple of str`*) – Which split to load. Valid values are 'train', 'valid' and 'test', corresponding to the training set (50,000 examples), the validation set (10,000 samples) and the test set (10,000 examples).

`filename` **= 'binarized_mnist.hdf5'**

7.3.6 CIFAR100

class `fuel.datasets.cifar100.`**`CIFAR100`**(*which_sets*, ***kwargs*)

Bases: `fuel.datasets.hdf5.H5PYDataset`

The CIFAR100 dataset of natural images.

This dataset is a labeled subset of the `80 million tiny images` dataset [TINY]. It consists of 60,000 32 x 32 colour images labelled into 100 fine-grained classes and 20 super-classes. There are 600 images per fine-grained class. There are 50,000 training images and 10,000 test images [CIFAR100].

The dataset contains three sources: - features: the images themselves, - coarse_labels: the superclasses 1-20, - fine_labels: the fine-grained classes 1-100.

Parameters **`which_sets`** (*`tuple of str`*) – Which split to load. Valid values are 'train' and 'test', corresponding to the training set (50,000 examples) and the test set (10,000 examples). Note that CIFAR100 does not have a validation set; usually you will create your own training/validation split using the *subset* argument.

`default_transformers` **= ((<class 'fuel.transformers.ScaleAndShift'>, [0.00392156862745098, 0], {'which_sources': (**

`filename` **= 'cifar100.hdf5'**

7.3.7 CIFAR10

class `fuel.datasets.cifar10.`**`CIFAR10`**(*which_sets*, ***kwargs*)

Bases: `fuel.datasets.hdf5.H5PYDataset`

The CIFAR10 dataset of natural images.

This dataset is a labeled subset of the `80 million tiny images` dataset [TINY]. It consists of 60,000 32 x 32 colour images in 10 classes, with 6,000 images per class. There are 50,000 training images and 10,000 test images [CIFAR10].

Parameters which_sets (*tuple of str*) – Which split to load. Valid values are 'train' and 'test', corresponding to the training set (50,000 examples) and the test set (10,000 examples). Note that CIFAR10 does not have a validation set; usually you will create your own training/validation split using the *subset* argument.

default_transformers = ((<class 'fuel.transformers.ScaleAndShift'>, [0.00392156862745098, 0], {'which_sources': (

filename = 'cifar10.hdf5'

7.3.8 IRIS

class `fuel.datasets.iris.`**`Iris`**(*which_sets*, ***kwargs*)

Bases: `fuel.datasets.hdf5.H5PYDataset`

Iris dataset.

Iris [IRIS] is a simple pattern recognition dataset, which consist of 3 classes of 50 examples each having 4 real-valued features each, where each class refers to a type of iris plant. It is accessible through the UCI Machine Learning repository [UCIIRIS].

Parameters which_sets (*tuple of str*) – Which split to load. Valid value is 'all' corresponding to 150 examples.

filename = 'iris.hdf5'

7.3.9 MNIST

class `fuel.datasets.mnist.`**`MNIST`**(*which_sets*, ***kwargs*)

Bases: `fuel.datasets.hdf5.H5PYDataset`

MNIST dataset.

MNIST (Mixed National Institute of Standards and Technology) [LBBH] is a database of handwritten digits. It is one of the most famous datasets in machine learning and consists of 60,000 training images and 10,000 testing images. The images are grayscale and 28 x 28 pixels large. It is accessible through Yann LeCun's website [LECUN].

Parameters which_sets (*tuple of str*) – Which split to load. Valid values are 'train' and 'test', corresponding to the training set (60,000 examples) and the test set (10,000 examples).

default_transformers = ((<class 'fuel.transformers.ScaleAndShift'>, [0.00392156862745098, 0], {'which_sources': (

filename = 'mnist.hdf5'

7.3.10 SVHN

class `fuel.datasets.svhn.`**`SVHN`**(*which_format*, *which_sets*, ***kwargs*)

Bases: `fuel.datasets.hdf5.H5PYDataset`

The Street View House Numbers (SVHN) dataset.

SVHN [SVHN] is a real-world image dataset for developing machine learning and object recognition algorithms with minimal requirement on data preprocessing and formatting. It can be seen as similar in flavor to MNIST [LBBH] (e.g., the images are of small cropped digits), but incorporates an order of magnitude more labeled data (over 600,000 digit images) and comes from a significantly harder, unsolved, real world problem (recognizing digits and numbers in natural scene images). SVHN is obtained from house numbers in Google Street View images.

Parameters

- **`which_format`** (*{1, 2}*) – SVHN format 1 contains the full numbers, whereas SVHN format 2 contains cropped digits.
- **`which_sets`** (*tuple of str*) – Which split to load. Valid values are 'train', 'test' and 'extra', corresponding to the training set (73,257 examples), the test set (26,032 examples) and the extra set (531,131 examples). Note that SVHN does not have a validation set; usually you will create your own training/validation split using the *subset* argument.

`default_transformers` = ((<class 'fuel.transformers.ScaleAndShift'>, [0.00392156862745098, 0], {'which_sources': (

`filename`

7.3.11 Text-based datasets

class `fuel.datasets.text.`**`TextFile`**(*files, dictionary, bos_token='<S>', eos_token='</S>', unk_token='<UNK>', level='word', preprocess=None*)

Bases: *`fuel.datasets.base.Dataset`*

Reads text files and numberizes them given a dictionary.

Parameters

- **`files`** (*list of str*) – The names of the files in order which they should be read. Each file is expected to have a sentence per line.
- **`dictionary`** (*str or dict*) – Either the path to a Pickled dictionary mapping tokens to integers, or the dictionary itself. At the very least this dictionary must map the unknown word-token to an integer.
- **`bos_token`** (*str or None, optional*) – The beginning-of-sentence (BOS) token in the dictionary that denotes the beginning of a sentence. Is `<S>` by default. If passed `None` no beginning of sentence markers will be added.
- **`eos_token`** (*str or None, optional*) – The end-of-sentence (EOS) token is `</S>` by default, see `bos_taken`.
- **`unk_token`** (*str, optional*) – The token in the dictionary to fall back on when a token could not be found in the dictionary.
- **`level`** (*'word' or 'character', optional*) – If 'word' the dictionary is expected to contain full words. The sentences in the text file will be split at the spaces, and each word replaced with its number as given by the dictionary, resulting in each example being a single list of numbers. If 'character' the dictionary is expected to contain single letters as keys. A single example will be a list of character numbers, starting with the first non-whitespace character and finishing with the last one.
- **`preprocess`** (*function, optional*) – A function which takes a sentence (string) as an input and returns a modified string. For example `str.lower` in order to lowercase the sentence before numberizing.

Examples

```
>>> with open('sentences.txt', 'w') as f:
...     _ = f.write("This is a sentence\n")
...     _ = f.write("This another one")
>>> dictionary = {'<UNK>': 0, '</S>': 1, 'this': 2, 'a': 3, 'one': 4}
>>> def lower(s):
```

```
...     return s.lower()
>>> text_data = TextFile(files=['sentences.txt'],
...                      dictionary=dictionary, bos_token=None,
...                      preprocess=lower)
>>> from fuel.streams import DataStream
>>> for data in DataStream(text_data).get_epoch_iterator():
...     print(data)
([2, 0, 3, 0, 1],)
([2, 0, 4, 1],)
```

example_iteration_scheme = None

get_data(*state=None*, *request=None*)

open()

provides_sources = ('features',)

7.3.12 Toy datasets

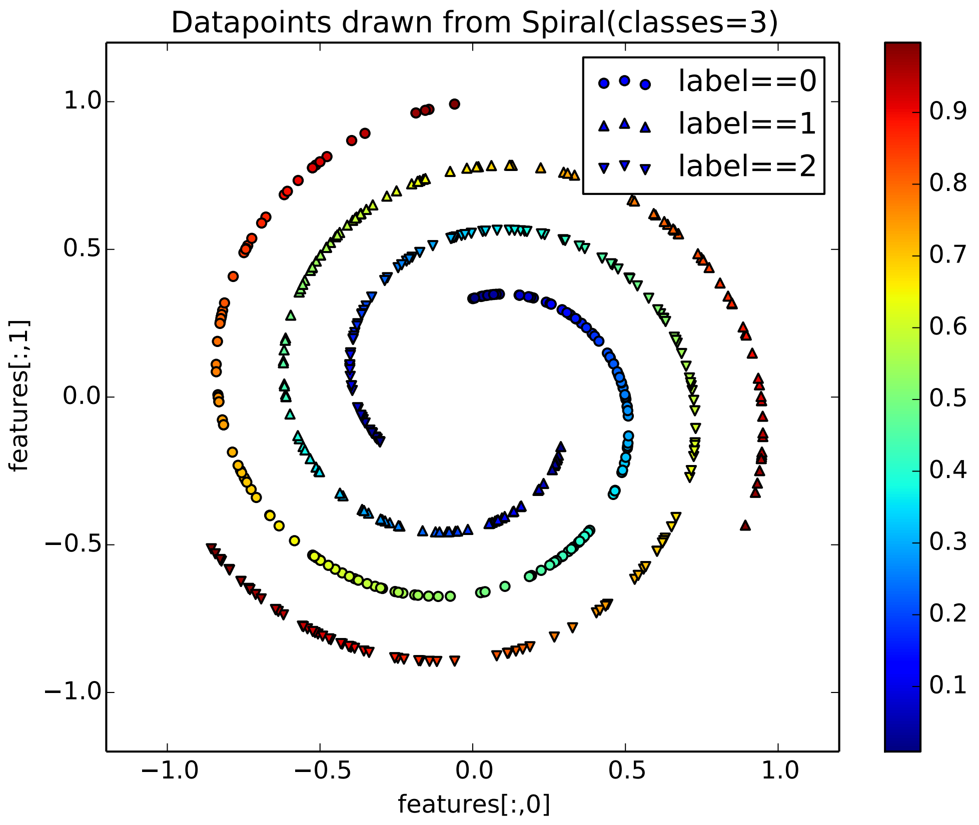
class fuel.datasets.toy.**Spiral**(*num_examples=1000*, *classes=1*, *cycles=1.0*, *noise=0.0*, ***kwargs*)

Bases: *fuel.datasets.base.IndexableDataset*

Toy dataset containing points sampled from spirals on a 2d plane.

The dataset contains 3 sources:

- features – the (x, y) position of the datapoints
- position – the relative position on the spiral arm
- label – the class labels (spiral arm)



Parameters

- **`num_examples`** (*`int`*) – Number of datapoints to create.
- **`classes`** (*`int`*) – Number of spiral arms.
- **`cycles`** (*`float`*) – Number of turns the arms take.
- **`noise`** (*`float`*) – Add normal distributed noise with standard deviation *noise*.

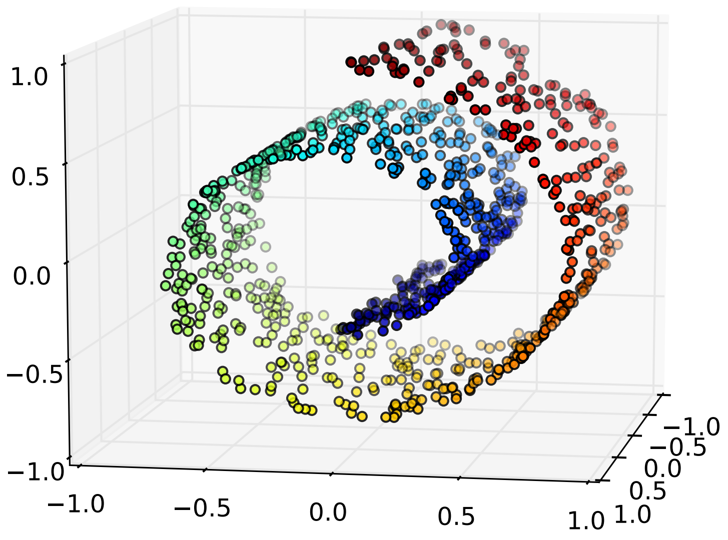
class `fuel.datasets.toy.`**`SwissRoll`**(*num_examples=1000*, *noise=0.0*, ***kwargs*)

Bases: *`fuel.datasets.base.IndexableDataset`*

Dataset containing points from a 3-dimensional Swiss roll.

The dataset contains 2 sources:

- features – the x, y and z position of the datapoints
- position – radial and z position on the manifold



Parameters

- **num_examples** (*int*) – Number of datapoints to create.
- **noise** (*float*) – Add normal distributed noise with standard deviation *noise*.

7.4 Downloaders

7.4.1 Base Classes

`fuel.downloaders.base.`**`default_downloader`**(*directory*, *urls*, *filenames*, *url_prefix=None*, *clear=False*)

Downloads or clears files from URLs and filenames.

Parameters

- **directory** (*str*) – The directory in which downloaded files are saved.
- **urls** (*list*) – A list of URLs to download.
- **filenames** (*list*) – A list of file names for the corresponding URLs.
- **url_prefix** (*str, optional*) – If provided, this is prepended to filenames that lack a corresponding URL.
- **clear** (*bool, optional*) – If *True*, delete the given filenames from the given directory rather than download them.

`fuel.downloaders.base.`**`download`**`(`*url*, *file_handle*, *chunk_size=1024*`)`
Downloads a given URL to a specific file.

Parameters

- **`url`** (`str`) – URL to download.
- **`file_handle`** (`file`) – Where to save the downloaded URL.

`fuel.downloaders.base.`**`ensure_directory_exists`**`(`*directory*`)`
Create directory (with parents) if does not exist, raise on failure.

Parameters **`directory`** (`str`) – The directory to create

`fuel.downloaders.base.`**`filename_from_url`**`(`*url*, *path=None*`)`
Parses a URL to determine a file name.

Parameters **`url`** (`str`) – URL to parse.

`fuel.downloaders.base.`**`progress_bar`**`(`**args*, ***kwds*`)`
Manages a progress bar for a download.

Parameters

- **`name`** (`str`) – Name of the downloaded file.
- **`maxval`** (`int`) – Total size of the download, in bytes.

7.4.2 Adult

`fuel.downloaders.adult.`**`fill_subparser`**`(`*subparser*`)`
Set up a subparser to download the adult dataset file.

The Adult dataset file *adult.data* and *adult.test* is downloaded from the UCI Machine Learning Repository [UCIADULT].

Parameters **`subparser`** (`argparse.ArgumentParser`) – Subparser handling the adult command.

7.4.3 CalTech 101 Silhouettes

`fuel.downloaders.caltech101_silhouettes.`**`fill_subparser`**`(`*subparser*`)`
Sets up a subparser to download the Silhouettes dataset files.

The following CalTech 101 Silhouette dataset files can be downloaded from Benjamin M. Marlin's website [MARLIN]: *caltech101_silhouettes_16_split1.mat* and *caltech101_silhouettes_28_split1.mat*.

Parameters **`subparser`** (`argparse.ArgumentParser`) – Subparser handling the *caltech101_silhouettes* command.

`fuel.downloaders.caltech101_silhouettes.`**`silhouettes_downloader`**`(`*size*, ***kwargs*`)`

7.4.4 Binarized MNIST

`fuel.downloaders.binarized_mnist.`**`fill_subparser`**`(`*subparser*`)`
Sets up a subparser to download the binarized MNIST dataset files.

The binarized MNIST dataset files (*binarized_mnist_{train,valid,test}.amat*) are downloaded from Hugo Larochelle's website [HUGO].

Parameters **subparser** (`argparse.ArgumentParser`) – Subparser handling the *binarized_mnist* command.

7.4.5 CIFAR100

`fuel.downloaders.cifar100.`**`fill_subparser`**(*subparser*)

Sets up a subparser to download the CIFAR-100 dataset file.

The CIFAR-100 dataset file is downloaded from Alex Krizhevsky's website [ALEX].

Parameters **subparser** (`argparse.ArgumentParser`) – Subparser handling the *cifar100* command.

7.4.6 CIFAR10

`fuel.downloaders.cifar10.`**`fill_subparser`**(*subparser*)

Sets up a subparser to download the CIFAR-10 dataset file.

The CIFAR-10 dataset file is downloaded from Alex Krizhevsky's website [ALEX].

Parameters **subparser** (`argparse.ArgumentParser`) – Subparser handling the *cifar10* command.

7.4.7 IRIS

`fuel.downloaders.iris.`**`fill_subparser`**(*subparser*)

Set up a subparser to download the Iris dataset file.

The Iris dataset file *iris.data* is downloaded from the UCI Machine Learning Repository [UCIIRIS].

Parameters **subparser** (`argparse.ArgumentParser`) – Subparser handling the iris command.

7.4.8 MNIST

`fuel.downloaders.mnist.`**`fill_subparser`**(*subparser*)

Sets up a subparser to download the MNIST dataset files.

The following MNIST dataset files are downloaded from Yann LeCun's website [LECUN]: *train-images-idx3-ubyte.gz*, *train-labels-idx1-ubyte.gz*, *t10k-images-idx3-ubyte.gz*, *t10k-labels-idx1-ubyte.gz*.

Parameters **subparser** (`argparse.ArgumentParser`) – Subparser handling the *mnist* command.

7.4.9 SVHN

`fuel.downloaders.svhn.`**`fill_subparser`**(*subparser*)

Sets up a subparser to download the SVHN dataset files.

The SVHN dataset files (*{train,test,extra}{.tar.gz,_32x32.mat}*) are downloaded from the official website [SVHNSITE].

Parameters **subparser** (`argparse.ArgumentParser`) – Subparser handling the *svhn* command.

`fuel.downloaders.svhn.`**`svhn_downloader`**`(`*which_format*, *directory*, *clear=False*`)`

7.5 Iteration schemes

class `fuel.schemes.`**`BatchScheme`**`(`*examples*, *batch_size*`)`

Bases: *`fuel.schemes.IterationScheme`*

Iteration schemes that return slices or indices for batches.

For datasets where the number of examples is known and easily accessible (as is the case for most datasets which are small enough to be kept in memory, like MNIST) we can provide slices or lists of labels to the dataset.

Parameters

- **`examples`** (*`int or list`*) – Defines which examples from the dataset are iterated. If list, its items are the indices of examples. If an integer, it will use that many examples from the beginning of the dataset, i.e. it is interpreted as range(examples)
- **`batch_size`** (*`int`*) – The request iterator will return slices or list of indices in batches of size *batch_size* until the end of *examples* is reached. Note that this means that the last batch size returned could be smaller than *batch_size*. If you want to ensure all batches are of equal size, then ensure len(*examples*) or *examples* is a multiple of *batch_size*.

`requests_examples` **= False**

class `fuel.schemes.`**`BatchSizeScheme`**

Bases: *`fuel.schemes.IterationScheme`*

Iteration scheme that returns batch sizes.

For infinite datasets it doesn’t make sense to provide indices to examples, but the number of samples per batch can still be given. Hence BatchSizeScheme is the base class for iteration schemes that only provide the number of examples that should be in a batch.

`requests_examples` **= False**

class `fuel.schemes.`**`ConcatenatedScheme`**`(`*schemes*`)`

Bases: *`fuel.schemes.IterationScheme`*

Build an iterator by concatenating several schemes’ iterators.

Useful for iterating through different subsets of data in a specific order.

Parameters **`schemes`** (*`list`*) – A list of `IterationSchemes`, whose request iterators are to be concatenated in the order given.

Notes

All schemes being concatenated must produce the same type of requests (batches or examples).

`get_request_iterator``()`

`requests_examples`

class `fuel.schemes.`**`ConstantScheme`**`(`*batch_size*, *num_examples=None*, *times=None*`)`

Bases: *`fuel.schemes.BatchSizeScheme`*

Constant batch size iterator.

This subset iterator simply returns the same constant batch size for a given number of times (or else infinitely).

Parameters

- **`batch_size`** (*`int`*) – The size of the batch to return.
- **`num_examples`** (*`int, optional`*) – If given, the request iterator will return *batch_size* until the sum reaches *num_examples*. Note that this means that the last batch size returned could be smaller than *batch_size*. If you want to ensure all batches are of equal size, then pass *times* equal to `num_examples / batch-size` instead.
- **`times`** (*`int, optional`*) – The number of times to return *batch_size*.

`get_request_iterator``()`

class `fuel.schemes.`**`IndexScheme`**`(`*examples*`)`

Bases: *`fuel.schemes.IterationScheme`*

Iteration schemes that return single indices.

This is for datasets that support indexing (like *`BatchScheme`*) but where we want to return single examples instead of batches.

`requests_examples` **= True**

class `fuel.schemes.`**`IterationScheme`**

Bases: `object`

An iteration scheme.

Iteration schemes provide a dataset-agnostic iteration scheme, such as sequential batches, shuffled batches, etc. for datasets that choose to support them.

`requests_examples`

bool

Whether requests produced by this scheme correspond to single examples (as opposed to batches).

Notes

Iteration schemes implement the *`get_request_iterator()`* method, which returns an iterator type (e.g. a generator or a class which implements the iterator protocol).

Stochastic iteration schemes should generally not be shared between different data streams, because it would make experiments harder to reproduce.

`get_request_iterator``()`

Returns an iterator type.

class `fuel.schemes.`**`SequentialExampleScheme`**`(`*examples*`)`

Bases: *`fuel.schemes.IndexScheme`*

Sequential examples iterator.

Returns examples in order.

`get_request_iterator``()`

class `fuel.schemes.`**`SequentialScheme`**`(`*examples, batch_size*`)`

Bases: *`fuel.schemes.BatchScheme`*

Sequential batches iterator.

Iterate over all the examples in a dataset of fixed size sequentially in batches of a given size.

Notes

The batch size isn't enforced, so the last batch could be smaller.

get_request_iterator()

class fuel.schemes.**ShuffledExampleScheme**(**args*, ***kwargs*)

Bases: *fuel.schemes.IndexScheme*

Shuffled examples iterator.

Returns examples in random order.

get_request_iterator()

class fuel.schemes.**ShuffledScheme**(**args*, ***kwargs*)

Bases: *fuel.schemes.BatchScheme*

Shuffled batches iterator.

Iterate over all the examples in a dataset of fixed size in shuffled batches.

Parameters sorted_indices (*bool, optional*) – If *True*, enforce that indices within a batch are ordered. Defaults to *False*.

Notes

The batch size isn't enforced, so the last batch could be smaller.

Shuffling the batches requires creating a shuffled list of indices in memory. This can be memory-intensive for very large numbers of examples (i.e. in the order of tens of millions).

get_request_iterator()

fuel.schemes.**cross_validation**(*scheme_class*, *num_examples*, *num_folds*, *strict=True*, ***kwargs*)

Return pairs of schemes to be used for cross-validation.

Parameters

- **scheme_class** (subclass of *IndexScheme* or *BatchScheme*) – The type of the returned schemes. The constructor is called with an iterator and ***kwargs* as arguments.
- **num_examples** (*int*) – The number of examples in the datastream.
- **num_folds** (*int*) – The number of folds to return.
- **strict** (*bool, optional*) – If *True*, enforce that *num_examples* is divisible by *num_folds* and so, that all validation sets have the same size. If *False*, the size of the validation set is returned along the iteration schemes. Defaults to *True*.

Yields fold (*tuple*) – The generator returns *num_folds* tuples. The first two elements of the tuple are the training and validation iteration schemes. If *strict* is set to *False*, the tuple has a third element corresponding to the size of the validation set.

7.6 Transformers

7.6.1 General transformers

class fuel.transformers.defaults.**ToBytes**(*stream*, ***kwargs*)

Bases: fuel.transformers.SourcewiseTransformer

Transform a stream of ndarray examples to bytes.

Notes

Used for retrieving variable-length byte data stored as, e.g. a uint8 ragged array.

`transform_source_batch`(*batch*, *_*)

`transform_source_example`(*example*, *_*)

`fuel.transformers.defaults.`**`rgb_images_from_encoded_bytes`**(*which_sources*)

`fuel.transformers.defaults.`**`uint8_pixels_to_floatX`**(*which_sources*)

7.6.2 Transformers for image

class `fuel.transformers.image.`**`ImagesFromBytes`**(*data_stream*, *color_mode='RGB'*, ***kwargs*)

Bases: `fuel.transformers.SourcewiseTransformer`

Load from a stream of bytes objects representing encoded images.

Parameters

- **`data_stream`** (instance of `AbstractDataStream`) – The wrapped data stream. The individual examples returned by this should be the bytes (in a *bytes* container on Python 3 or a *str* on Python 2) comprising an image in a format readable by PIL, such as PNG, JPEG, etc.
- **`color_mode`** (`str, optional`) – Mode to pass to PIL for color space conversion. Default is RGB. If *None*, no coercion is performed.

Notes

Images are returned as NumPy arrays converted from PIL objects. If there is more than one color channel, then the array is transposed from the *(height, width, channel)* dimension layout native to PIL to the *(channel, height, width)* layout that is pervasive in the world of convolutional networks. If there is only one color channel, as for monochrome or binary images, a leading axis with length 1 is added for the sake of uniformity/predictability.

This SourcewiseTransformer supports streams returning single examples as *bytes* objects (*str* on Python 2.x) as well as streams that return iterables containing such objects. In the case of an iterable, a list of loaded images is returned.

`transform_source_batch`(*batch*, *source_name*)

`transform_source_example`(*example*, *source_name*)

class `fuel.transformers.image.`**`MinimumImageDimensions`**(*data_stream*, *minimum_shape*, *resample='nearest'*, ***kwargs*)

Bases: `fuel.transformers.SourcewiseTransformer`, `fuel.transformers.ExpectsAxisLabels`

Resize (lists of) images to minimum dimensions.

Parameters

- **`data_stream`** (instance of `AbstractDataStream`) – The data stream to wrap.
- **`minimum_shape`** (`2-tuple`) – The minimum *(height, width)* dimensions every image must have. Images whose height and width are larger than these dimensions are passed through as-is.

- **`resample`** (*`str, optional`*) – Resampling filter for PIL to use to upsample any images requiring it. Options include 'nearest' (default), 'bilinear', and 'bicubic'. See the PIL documentation for more detailed information.

Notes

This transformer expects stream sources returning individual images, represented as 2- or 3-dimensional arrays, or lists of the same. The format of the stream is unaltered.

`transform_source_batch`(*batch*, *source_name*)

`transform_source_example`(*example*, *source_name*)

class `fuel.transformers.image.`**`RandomFixedSizeCrop`**(*data_stream*, *window_shape*, ***kwargs*)

Bases: `fuel.transformers.SourcewiseTransformer`, `fuel.transformers.ExpectsAxisLabels`

Randomly crop images to a fixed window size.

Parameters

- **`data_stream`** (`AbstractDataStream`) – The data stream to wrap.
- **`window_shape`** (*`tuple`*) – The *(height, width)* tuple representing the size of the output window.

Notes

This transformer expects to act on stream sources which provide one of

- Single images represented as 3-dimensional ndarrays, with layout *(channel, height, width)*.
- Batches of images represented as lists of 3-dimensional ndarrays, possibly of different shapes (i.e. images of differing heights/widths).
- Batches of images represented as 4-dimensional ndarrays, with layout *(batch, channel, height, width)*.

The format of the stream will be un-altered, i.e. if lists are yielded by *data_stream* then lists will be yielded by this transformer.

`transform_source_batch`(*source*, *source_name*)

`transform_source_example`(*example*, *source_name*)

7.6.3 Transformers for text

class `fuel.transformers.text.`**`NGrams`**(*ngram_order*, *data_stream*, *target_source='targets'*, ***kwargs*)

Bases: `fuel.transformers.Transformer`

Return n-grams from a stream.

This data stream wrapper takes as an input a data stream outputting sentences. From these sentences n-grams of a fixed order (e.g. bigrams, trigrams, etc.) are extracted and returned. It also creates a `targets` data source. For each example, the target is the word immediately following that n-gram. It is normally used for language modeling, where we try to predict the next word from the previous *n* words.

Parameters

- **`ngram_order`** (*`int`*) – The order of the n-grams to output e.g. 3 for trigrams.

- **data_stream** (*DataStream* instance) – The data stream providing sentences. Each example is assumed to be a list of integers.
- **target_source** (*str, optional*) – This data stream adds a new source for the target words. By default this source is 'targets'.

get_data(*request=None*)

Fuel is a data pipeline framework which provides your machine learning models with the data they need. It is planned to be used by both the Blocks and Pylearn2 neural network libraries.

- Fuel allows you to easily read different types of data (NumPy binary files, CSV files, HDF5 files, text files) using a single interface which is based on Python's iterator types.
- Provides a a series of wrappers around frequently used datasets such as MNIST, CIFAR-10 (vision), the One Billion Word Dataset (text corpus), and many more.
- Allows you iterate over data in a variety of ways, e.g. in order, shuffled, sampled, etc.
- Gives you the possibility to process your data on-the-fly through a series of (chained) transformation procedures. This way you can whiten your data, noise, rotate, crop, pad, sort or shuffle, cache it, and much more.
- Is pickle-friendly, allowing you to stop and resume long-running experiments in the middle of a pass over your dataset without losing any training progress.

Warning: Fuel is a new project which is still under development. As such, certain (all) parts of the framework are subject to change. The last stable (but possibly outdated) release can be found in the `stable` branch.

Tip: That said, if you are interested in using Fuel and run into any problems, feel free to ask your question on the mailing list. Also, don't hesitate to file bug reports and feature requests by making a GitHub issue.

CHAPTER 8

Motivation

Fuel was originally factored out of the Blocks framework in the hope of being useful to other frameworks such as Pylearn2 as well. It shares similarities with the skdata package, but with a much heavier focus on data iteration and processing.

CHAPTER 9

Quickstart

The best way to get started with Fuel is to have a look at the overview documentation section.

CHAPTER 10

Indices and tables

- genindex
- modindex

Bibliography

[DBN] Ruslan Salakhutdinov and Iain Murray, *On the Quantitative Analysis of Deep Belief Networks*, Proceedings of the 25th international conference on Machine learning, 2008, pp. 872-879.

[SVHN] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, Andrew Y. Ng. *Reading Digits in Natural Images with Unsupervised Feature Learning*, NIPS Workshop on Deep Learning and Unsupervised Feature Learning, 2011.

[SVHNSITE] http://ufldl.stanford.edu/housenumbers/

[CSMG14] Ciprian Chelba, Tomas Mikolov, Mike Schuster, Qi Ge, and Thorsten Brants, *One Billion Word Benchmark for Measuring Progress in Statistical Language Modeling*, *arXiv:1312.3005 [cs.CL] <http://arxiv.org/abs/1312.3005>*.

[LBBH] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner, *Gradient-based learning applied to document recognition*, Proceedings of the IEEE, November 1998, 86(11):2278-2324.

[TINY] Antonio Torralba, Rob Fergus and William T. Freeman, *80 million tiny images: a large dataset for non-parametric object and scene recognition*, Pattern Analysis and Machine Intelligence, IEEE Transactions on 30.11 (2008): 1958-1970.

[CIFAR100] Alex Krizhevsky, *Learning Multiple Layers of Features from Tiny Images*, technical report, 2009.

[CIFAR10] Alex Krizhevsky, *Learning Multiple Layers of Features from Tiny Images*, technical report, 2009.

[IRIS] Ronald A. Fisher, *The use of multiple measurements in taxonomic problems*, Annual Eugenics, 7, Part II, 179-188, September 1936.

[UCIIRIS] https://archive.ics.uci.edu/ml/datasets/Iris

[LECUN] http://yann.lecun.com/exdb/mnist/

[UCIADULT] https://archive.ics.uci.edu/ml/datasets/Adult

[MARLIN] https://people.cs.umass.edu/~marlin/data.shtml

[HUGO] http://www.cs.toronto.edu/~larocheh/public/datasets/ binarized_mnist/binarized_mnist_{train,valid,test}.amat

[ALEX] http://www.cs.toronto.edu/~kriz/cifar-100-python.tar.gz

Index