Pylearn2 Documentation

Release dev

LISA lab, University of Montreal

May 08, 2015

Contents

Pylearn2 is still undergoing rapid development. Don't expect a clean road without bumps! If you find a bug please write to pylearn-dev@googlegroups.com. If you're a Pylearn2 developer and you find a bug, please write a unit test for it so the bug doesn't come back!

Pylearn2 is a machine learning library. Most of its functionality is built on top of Theano. This means you can write Pylearn2 plugins (new models, algorithms, etc) using mathematical expressions, and Theano will optimize and stabilize those expressions for you, and compile them to a backend of your choice (CPU or GPU).

CHAPTER 1

Pylearn2 Vision

- Researchers add features as they need them. We avoid getting bogged down by too much top-down planning in advance.
- A machine learning toolbox for easy scientific experimentation.
- All models/algorithms published by the LISA lab should have reference implementations in Pylearn2.
- Pylearn2 may wrap other libraries such as scikit-learn when this is practical
- Pylearn2 differs from scikit-learn in that Pylearn2 aims to provide great flexibility and make it possible for a researcher to do almost anything, while scikit-learn aims to work as a "black box" that can produce good results even if the user does not understand the implementation
- Dataset interface for vector, images, video, ...
- Small framework for all what is needed for one normal MLP/RBM/SDA/Convolution experiments.
- Easy reuse of sub-component of Pylearn2.
- Using one sub-component of the library does not force you to use / learn to use all of the other sub-components if you choose not to.
- Support cross-platform serialization of learned models.
- Remain approachable enough to be used in the classroom (IFT6266 at the University of Montreal).

CHAPTER 2

Download and installation

There is no PyPI download yet, so Pylearn2 cannot be installed using e.g. `pip`. You must check out the bleeding-edge/development version from GitHub using:

```
git clone git://github.com/lisa-lab/pylearn2.git
```

To make Pylearn2 available in your Python installation, run the following command in the top-level `pylearn2` directory (which should have been created by the previous command):

```
python setup.py develop
```

You may need to use `sudo` to invoke this command with administrator privileges. If you do not have such access (or would rather not add Pylearn2 to the global *site-packages* for whatever reason), you can install the relevant links inside the user site-packages directory by issuing the command:

```
python setup.py develop --user
```

This command will also compile the Cython extensions required for e.g. `pylearn2.train_extensions.window_flip`.

Alternatively, you can make Pylearn2 available by adding the installation directory to your `PYTHONPATH` environment variable, but note that changing your `PYTHONPATH` alone won't compile the Cython extensions (you will need to make sure the extension `.so` files are built and accessible within the source tree, e.g. with `python setup.py build_ext --inplace`).

2.1 Data path

For some tutorials and tests you will also need to set your `PYLEARN2_DATA_PATH` variable. On Linux, the best way to do this is to add a line to your `~/.bashrc` file:

```
export PYLEARN2_DATA_PATH=/data/lisa/data
```

Note that this is only an example, and if you are not in the LISA lab, you will need to choose a directory path that is valid on your filesystem. Simply choose a path where it will be convenient for you to store datasets for use with Pylearn2.

2.2 Other methods

2.2.1 Vagrant (any OS)

Pylearn2 in a box uses Vagrant to easily create a new VM installed with Pylearn2 and the necessary packages.

1. Download and install Vagrant http://www.vagrantup.com/
2. Download and install Virtual Box and VirtualBox Extension Pack https://www.virtualbox.org/wiki/Downloads
3. Download the Vagrant scripts from Pylearn2 in a box
4. Start the VM by running `vagrant up` in the directory from step 3

CHAPTER 3

Dependencies

- Theano and its dependencies are required to use Pylearn2. Since pylearn2 is under rapid development some of its features might depend on the bleeding-edge version of Theano.
- PyYAML is required for most functionality.
- PIL is required for some image-related functionality.
- matplotlib is required for some plotting functionality.
- **Some dependencies are optional:**
  - Pylearn2 includes code for accessing several standard datasets, such as MNIST and CIFAR-10. However, if you wish to use one of these datasets, you must download the dataset itself manually.
  - The original Pylearn project is required for loading some datasets, such as the Unsupervised and Transfer Learning Challenge datasets
  - Some features (SVMs) depend on scikit-learn.
  - k-means depends on milk.
  - Reading SVHN dataset depends on pytables.

CHAPTER 4

License and Citations

Pylearn2 is released under the 3-claused BSD license, so it may be used for commercial purposes. The license does not require anyone to cite Pylearn2, but if you use Pylearn2 in published research work we encourage you to cite this article:

- Ian J. Goodfellow, David Warde-Farley, Pascal Lamblin, Vincent Dumoulin, Mehdi Mirza, Razvan Pascanu, James Bergstra, Frédéric Bastien, and Yoshua Bengio. "Pylearn2: a machine learning research library". *arXiv preprint arXiv:1308.4214* (BibTeX)

Pylearn2 is primarily developed by academics, and so citations matter a lot to us. As an added benefit, you increase Pylearn2's exposure and potential user (and developer) base, which is to the benefit of all users of Pylearn2. Thanks in advance!

CHAPTER 5

Documentation

Roughly in order of what you'll want to check out:

- *Quick-start example* – Learn the basics via an example.
- *YAML for Pylearn2* – A tutorial on YAML tags employed by Pylearn2.
- *IPython Notebook Tutorials* – At this point, you might want to work through the ipython notebooks in the "scripts/tutorials" directory.
- A First Experiment with Pylearn2 – A brief introduction to running experiments.
- Monitoring Experiments in Pylearn2 – An overview of monitoring experiments.
- *Your models in Pylearn2* – A tutorial on porting Theano code to Pylearn2
- *Features* – A list of features available in the library.
- *Overview* – A detailed but high-level overview of how Pylearn2 works. This is the place to start if you want to really learn the library.
- *Library Documentation* – Documentation of the library modules.
- *Working with computer clusters* – The tools we use at LISA for running Pylearn2 jobs on HPC clusters.
- *Working with large datasets in Pylearn2* – A guide on how to deal with large datasets.
- *Pylearn2 Vision* – Some more detailed elaboration of some points of the Pylearn2 vision.
- *F.A.Q.* – Please read the FAQ section before posting to mailing-lists.

CHAPTER 6

Community

- Register and post to pylearn-users for general inquiries and support questions or if you want to talk to other users.
- Register and post to pylearn-dev if you want to talk to the developers. We don't bite.
- Register to pylearn2-github if you want to receive an email for all changes to the GitHub repository.
- Register to theano-buildbot if you want to receive our daily buildbot email. This is the buildbot for Pylearn2, Theano, Pylearn and the Deep Learning Tutorial.
- Ask/view questions/answers about machine learning in general at metaoptimize/qa/tags/theano (it's like stack overflow for machine learning)
- We use the github issues to stay organized.
- Come visit us in Montreal! Most of the developers are students in the LISA group at the University of Montreal.

CHAPTER 7

Developer

- Register to everything listed in the Community section above
- Follow the LISA lab coding style guidelines: http://deeplearning.net/software/pylearn/v2_planning/API_coding_style.
- *Pylearn2 API Change Best Practices Guide* – the best practices guide you should follow when changing any API in the library
- *Developer Start Guide* – how to contribute code to Pylearn2
- *Pylearn2 Pull Request Checklist* – Things you should check your pull request for before review.
- *Data specifications, spaces, and sources* – the interface different elements use to request and provide data

7.1 Library Documentation

7.1.1 Datasets

Contents

Dataset Base Classes

Dataset

Dense Design Matrix

List of Datasets

MNIST

MNIST+

CIFAR10

CIFAR100

Street View House Numbers (SVHN)

STL 10

Adult

Binarizer

COS dataset

CSV dataset

Hepatitis

Icml07

Iris

Matlab Dataset

Norb

Small Norb

Npy dataset

OCR

Sparse dataset

h5py dataset

Toronto Face Dataset

Contents

Models Base Class

K-means

MLP

Principal Component Analysis

RBM

DBM

Ising

Auto Encoder

Maxout

Differentiable sparse coding

GSN

Independent multiclass logistic

Local coordinate coding

Multivariate normal distribution

Normalized EBM

S3C

Softmax Regression

Sparse Autoencoder

SVM

7.1.3 Corruption

7.1.4 Training

Contents

pylearn2.train.py

pylearn2.scripts.train.py

See *Train*.

Training Algorithms

Base Class

Default

Learning rule

Stochastic Gradient Descent

Batch Gradient Descent

Train Extensions

Basic

Best Params

Window Flip

7.1.5 Costs

Basic

MLP

Basic

Dropout

Missing target cost

Auto Encoder

DBM

EBM

GSN

7.1.6 Monitoring

Monitor

7.1.7 Linear Transform

Contents

Intro

Functions

2D Convolution

2D Convolution C01B

Matrix Multiplication

Linear Transformation

Local C01B

7.1.8 Cuda-Convnet

Contents

Intro

These are wrappers around some of the GPU code from Alex Krizhevsky cuda-convnet project. http://code.google.com/p/cuda-convnet/

Sander Dieleman wrote an excellent blog post that describes how to use this module without the rest of Pylearn2.

Modules

Convolution

Pool

Response Normalization

7.1.9 Scripts

Contents

Monitoring

Diff Monitor

Num Parameters

Plot Monitor

Print Channel Doc

Print model

Summarize Model

Visualization

Show binocular greyscale examples

Show Examples

Show Weights

Other

Find GPU fields

GPU to CPU pkl

Pkl inspector

Train

Predict

7.1.10 Configuration

YAML parser

Old configuration

7.1.11 Costs

Basic

MLP

Basic

Dropout

Contents

Basic

7.1.21 Space

Contents

Basic

7.1.22 Miscellaneous

Contents

Blocks

RBM tools

7.2 Pylearn2 API Change Best Practices Guide

API changes are any changes that affect the way a piece of code interacts with the outside world. This means changing the name of any class, function, method, parameter name. It can also mean changing a default argument value, or changing the behavior that results from setting an argument to a certain value. Because API changes require the outside world to adapt to changes in pylearn2, they should be done as rarely as possible, and with great care. As of this writing, 145 people are watching pylearn2 on github. Pylearn2 is used to store reproducible scientific results, and has been used to host five different contests on kaggle.com. Breaking interfaces in pylearn2 will upset people all over the internet. Unfortunately, given pylearn2's mission of being a cutting-edge research library, new research directions will often require changes to pylearn2's interface. This guide explains how to change APIs while doing the least damage possible.

- E-mail pylearn-dev@googlegroups.com about proposed changes before you start working on them. This will give the other users of pylearn2 the opportunity to see possible problems your changes might cause, and help you figure out the safest possible way to accomplish what you want to with the API change.
- Whenever possible, make the old version of the interface still work for 6 months before it is phased out.
  - You should detect when someone uses the old interface, and use

    warnings.warn("<Old interface element> is deprecated. Use " "<new interface element> instead. <Old interface element> will be removed" "on or after <date 6 months from now>", stacklevel=2)

    to warn that the interface is going to change.
  - If the interface change doesn't cause any serious behavior changes (i.e., you're just not asking for some metadata that is no longer required to accomplish the task) then it's OK to just issue the warning. But if the change that takes place after the six month warning will alter numerical results, or cause a crash if someone uses the old interface, you should take the following steps to update everything in the library to the new interface:
    * Put a "raise AssertionError()" before all of your new warnings, and run the unit tests. Fix all tests that fail because they would trigger your deprecation warning.
    * Use "grep -r" to find all instances of the name of the thing you're changing, and make sure that they all use the right interface. Sadly, not everything in pylearn2 is unit tested, so just running the tests might miss some cases.
    * Remove the "raise AssertionError()" statements before sending your pull request. However, you must leave the warnings in, because clients of the library might (and probably are) still be using the old interface.
  - If you're changing the name of a parameter of a function / method, keep the new name, add the old name, and have both default to None. Inside the function, raise a warning if the old name is specified. If both the old name and the new name are specified, raise a warning if both are set to the same value. Raise an error if both are set, and they're set to different values. If both are set to None, fill in the default value. If you are also planning to change the default after the end of the deprecation warning period, add a warning whenever the default is filled in.

    Example:

```
# Suppose you have a function

def my_func(my_arg = 1):

# and you want to change it to

def my_func(arg = 2):
```

```
        # You should start your function off like this:

        def my_func(my_arg = None, arg = None):

            if my_arg is not None:
                warnings.warn("my_arg is deprecated. Use arg instead. my_arg will be remo
                    "on or after October 18, 2013")
                if arg is not None:
                    if arg == my_arg:
                        warnings.warn("You are specifying both arg and my_arg. They are a
                         "You are setting them both to the same thing, so it's OK, but yo
                         "probably switch to specifying only arg")
                    else:
                        raise ValueError("my_arg and arg are aliases, but you set them to
                         "values")
                arg = my_arg

            if arg is None:
                warnings.warn("You are using the default value of arg. This default will
                 "after October 18, 2013.")
                arg = 2
```

- Make sure there are unit tests for the functionality you're changing. If you have to write a lot of logic to handle warnings, changes to default values, etc. write unit tests to make sure the old interface still works. This includes making sure that the behavior when you pass no arguments is the same (i.e., make sure you didn't change the default behavior by accident).

7.3 Working with computer clusters

Pylearn2 doesn't have any features explicitly designed to run large batches of experiments on computer clusters, but other related tools exist that enable you to run your Pylearn2 experiments in that way.

7.3.1 Jobdispatch

Script that allows you to submit jobs in an standard interface to multiple cluster that use different scheduler.

7.3.2 Jobman

Allow to make a db of all wanted jobs to run and manage there executions(ensure all ended correctly, ...). It support grid search and random search out of the box. The experiment descriptions are compatible with pylearn2's yaml format.

There is the Hyperplot project that allow to produce graphs and/or tables with jobman jobs: https://github.com/simlmx/hyperplot

Use postgres as a back-end database.

7.3.3 Hyperopt

James project that allow to do "smart" hyper-parameter search. It also allow to manage jobs similar to jobman, but use mangodb as a back-end batabase.

It support atomic reservation of jobs.

For distributed optimization there are a few commandline utilities of interest: * hyperopt-mongo-search controls an optimization experiment * hyperopt-mongo-worker runs on worker nodes and polls a mongodb for experiments that need to be run. * hyperopt-mongo-show wraps a couple of visualization strategies of running experiments.

There is documentation coming along here: https://github.com/jaberg/hyperopt/wiki

Fred doesn't have time to work on a switch from jobman to hyperopt to manage jobs. You can do try it if you want, but Fred won't be able to help much.

7.4 Features

Pylearn2 contains at least the following features. This documentation often gets out of date, so it probably has even more than this!

If there is something missing that you would like to have, write to pylearn-dev@googlegroups.com. We'll tell you if we've added it and forgotten to add it to the list here. Or if we haven't added that feature yet, we'll either add it or give you advice about how to do it yourself.

- **Training algorithms**
  - A "default training algorithm" that asks the model to train itself
  - **Stochastic gradient descent, with extensions including**
    - Learning rate decay
    - Momentum
    - Polyak averaging
    - Early stopping
    - A simple framework for adding your own extensions
  - Batch gradient descent with line searches
  - Nonlinear conjugate gradient descent (with line searches)
- **Model Estimation Criteria**
  - Score Matching
  - Denoising Score Matching
  - Noise-Contrastive Estimation
  - Cross-entropy
  - Log-likelihood

- **Models**
  - Autoencoders, including Contractive and Denoising Autoencoders
  - **RBMs, including gaussian and ssRBM. Varying levels of integration into** the full framework.
  - k-means
  - Local Coordinate Coding
  - Maxout networks
  - PCA
  - Spike-and-Slab Sparse coding
  - **SVMs (we provide a wrapper around scikit-learn that makes it easy to** train a multi-class svm on dense training data in a memory efficient way, which doesn't always happen using scikit-learn directly)
  - **Partial implementation of DBMs (contact Ian Goodfellow if you would like** to complete it)
- **Datasets:**
  - MNIST, MNIST with background and rotations
  - STL-10
  - CIFAR-10, CIFAR-100
  - NIPS Workshops 2011 Transfer Learning Challenge
  - UTLC
  - NORB
  - Toronto Faces Dataset
- **Dataset pre-processing**
  - Contrast normalization
  - ZCA whitening
  - Patch extraction (for implementing convolution-like algorithms)
  - The Coates+Lee+Ng CIFAR processing pipeline
- **Miscellaneous algorithms and utilities:**
  - AIS
  - Weight visualization for single layer networks
  - **Can plot learning curves showing how user-configured quantities** change during learning

7.5 Internal Documentation

7.5.1 Developer Start Guide

To get up to speed, you'll need to

- Learn some non-basic Python to understand what's going on in some of the trickier files (like tensor.py).
- Go through the NumPy documentation.
- Learn to write reStructuredText for epydoc and Sphinx.
- Learn about how unittest and nose work

Accounts

To obtain developer access: register with GitHub and create a fork of Pylearn2.

Coding style

See the coding style guideline here.

We do not plan to change all existing code to follow this coding style, but as we modify the code, we update it accordingly.

Documentation

See the documentation guideline *here*.

Mailing list

See the Pylearn2 main page for the pylearn-dev, theano-buildbot and pylearn2-github mailing list. They are useful to Pylearn2 contributors.

Git config

```
git config --global user.email you@yourdomain.example.com
git config --global user.name "Your Name Comes Here"
```

Typical development workflow

Clone your fork locally with

```
git clone git@github.com:your_github_login/pylearn2.git
```

and add a reference to the 'central' Pylearn2 repository with

```
git remote add central git://github.com/lisa-lab/pylearn2.git
```

When working on a new feature in your own fork, start from an up-to-date copy of the trunk:

```
git fetch central
git checkout -b my_shiny_feature central/master
```

Once your code is ready for others to review, push your branch to your github fork:

```
git push -u origin my_shiny_feature
```

then go to your fork's github page on the github website, select your feature branch and hit the "Pull Request" button in the top right corner. If you don't get any feedback, bug us on the pylearn-dev mailing list.

When the your pull request have been merged, you can delete the branch from the github list of branch. That is usefull to don't have too many that stay there!

```
git push origin :my_shiny_feature
```

You can keep you local repo up to date with central/master with those commands:

```
git checkout master
git fetch central
git merge central/master
```

If you want to fix a commit done in a pull request(i.e. fix small typo) to keep the history clean, you can do it like this:

```
git checkout branch
git commit --amend
git push -u origin  my_shiny_feature:my_shiny_feature
```

Coding Style Auto Check

See the coding style guideline here. The principal thing to know is that we follow the pep8 coding style.

We use git hooks provided in the project pygithooks to validate that commits respect pep8. This happens when each user commits, not when we push/merge to the Pylearn2 repository. Github doesn't allow us to have code executed when we push to the repository. So we ask all contributors to use those hooks.

For historic reason, we currently don't have all files respecting pep8. We decided to fix everything incrementally. So not all files respect it now. So we strongly suggest that you use the "increment" pygithooks config option to have a good workflow. See the pygithooks main page for how to set it up for Pylearn2 and how to enable this option.

Cleaning up history

Sometimes you may have commits in your feature branch that are not needed in the final pull request. There is a page that talks about this. In summary:

- Commits to the trunk should be a lot cleaner than commits to your feature branch; not just for ease of reviewing but also because intermediate commits can break blame (the bisecting tool).
- *git merge –squash* will put all of the commits from your feature branch into one commit.
- There are other tools that are useful if your branch is too big for one squash.

To checkout another user branch in his repo:

```
git remote add REPO_NAME HIS_REPO_PATH
git checkout -b LOCAL_BRANCH_NAME REPO_NAME/REMOVE_BRANCH_NAME
```

You can find move information and tips in the numpy development page.

Details about PYTHONPATH

`$PYTHONPATH` should contain a ":"-separated list of paths, each of which contains one or several Python packages, in the order in which you would like Python to search for them. If a package has sub-packages of interest to you, do **not** add them to `$PYTHONPATH`: it is not portable, might shadow other packages or short-circuit important things in its `__init__`.

It is advisable to never import Pylearn2's files from outside Pylearn2 itself (this is good advice for Python packages in general). Use `from pylearn2 import module` instead of `import module`. `$PYTHONPATH` should only contain paths to complete packages.

When you install a package, only the package name can be imported directly. If you want a sub-package, you must import it from the main package. That's how it will work in 99.9% of installs because it is the default. Therefore, if you stray from this practice, your code will not be portable. Also, some ways to circumvent circular dependencies might make it so you have to import files in a certain order, which is best handled by the package's own `__init__.py`.

More instructions

Once you have completed these steps, you should run the by running `nosetests` from your checkout directory.

All tests should pass. If some test fails on your machine, you are encouraged to tell us what went wrong on the pylearn-dev mailing list.

To update your library to the latest revision, you should have a branch that tracks the main trunk. You can add one with:

```
git fetch central
git branch trunk central/master
```

Once you have such a branch, in order to update it, do:

```
git checkout trunk
git pull
```

Keep in mind that this branch should be "read-only": if you want to patch Pylearn2, do it in another branch like described above.

Optional

You can instruct git to do color diff. For this, you need to add those lines in the file ~/.gitconfig

```
[color]
   branch = auto
   diff = auto
   interactive = auto
   status = auto
```

Nightly test

Each night we execute all the unit tests automatically. The result is sent by email to the theano-buildbot mailing list.

For more detail, see *see*.

To run all the tests with the same configuration as the buildbot, run this script:

```
pylearn2/misc/do_nightly_build
```

This function accepts arguments that it forward to nosetests. You can run only some tests or enable pdb by giving the equivalent nosetests parameters.

7.5.2 Data specifications, spaces, and sources

Data specifications, often called `data_specs`, are used as a specification for requesting and providing data in a certain format across different parts of a Pylearn2 experiment.

A `data_specs` is a `(space, source)` pair, where `source` is an identifier or the data source or sources required (for instance, inputs and targets), and `space` is an instance of `space.Space` representing the *format* of these data (for instance, a vector, a 3D tensor representing an RGB image, or a one-hot vector).

The main use of `data_specs` is to request data from a `datasets.Dataset` object, via an iterator. Various objects can request data this way: models, costs, monitoring channels, training algorithms, even some datasets that perform a transformation on data.

7.5.3 The `Space` object

A `Space` represents a way in which a mini-batch of data can be formatted. For instance, a batch of RGB images (each of shape `(rows, columns)`) can be represented in different ways, for instance:

- as a matrix where each row corresponds to a different image, and is of length `rows * columns * 3`: the corresponding space would be a `space.VectorSpace`, more precisely `VectorSpace(dim=(rows * columns * 3))`;
- as a 4-dimensional tensor, where rows, columns, and channels (here: red, green, and blue) are different axes: the corresponding space would be a `space.Conv2DSpace`. Theano convolutions prefer that tensor to have shape `(batch_size, channels, rows, columns)`, which corresponds to `Conv2DSpace(shape=(rows, columns), num_channels=3, axes=('b', 'c', 0, 1))`;
- as a 4-dimensional tensors with a different shape: for instance, cuda-convnet prefers `(channels, rows, columns, batch_size)`: the space would be `Conv2DSpace(shape=(rows, columns), num_channels=3, axes=('c', 0, 1, 'b'))`.

Spaces can be either elementary, representing one mini-batch from one source of data, such as `VectorSpace` and `Conv2DSpace` mentioned above, or *composite* (`space.CompositeSpace`), representing the aggregation of several sources of data (some of these may in turn be aggregations of sources). A mini-batch for an elementary space will usually be a NumPy `ndarray`, whereas a mini-batch for a `CompositeSpace` will be a Python tuple of elementary (or composite) mini-batches.

Notable methods of the `Space` class are:

- `Space.`**`make_theano_batch`**`()`

  creates a Theano Variable (or tuple of Theano Variable in the case of `CompositeSpace`) representing a *symbolic* mini-batch of data. For instance, `VectorSpace(...).make_theano_batch(...)` will essentially call `theano.tensor.matrix()`.

- `Space.`**`validate`**`(`*batch*`)`

  will check that symbolic variable `batch` can correctly represent a mini-batch of data for the corresponding space. For instance, `VectorSpace(...).validate(theano.tensor.matrix())` will work, but `VectorSpace(...).validate(theano.tensor.vector())` will raise an exception.

- `Space.`**`np_validate`**`(`*batch*`)`

  (`np` stands for NumPy) is similar, but operates on a mini-batch of numeric data, rather than on a symbolic variable. This enables more checks to be performed. For instance, `VectorSpace(dim=3).validate(np.zeros((4, 3)))` will work, because it correctly describes a mini-batch of 4 samples of dimension 3, but `VectorSpace(dim=4).validate(np.zeros((4, 3)))` will raise an exception.

- `Space.`**`format_as`**`(`*batch*, *space*`)`

  and

  `Space.`**`np_format_as`**`(`*batch*, *space*`)`

  are the way we can convert data from their original space into the destination `space`. `format_as` operates on a symbolic `batch`, and returns a symbolic expression of the newly-formatted data, whereas `np_format_as` operates on a numeric batch, and returns numeric data. This formatting can happen between different instances of the same `Space` class, for instance, converting between two instances of `Conv2DSpace` with different `axes` amounts to correctly transpose the `batch`. It can also

happen between different subclasses of `Space`, for instance, converting between a `VectorSpace` and `Conv2DSpace` of compatible shape involves reshaping and transposition of the data.

7.5.4 Sources

Sources are simple identifiers that specify *which* data should be returned, whereas spaces specify *how* that data should be formatted.

An elementary source is identified by a Python string. For instance, the most used sources are `'features'` and `'targets'`. `'features'` usually denotes the part of the data that models will use as input, and `'targets'`, for labeled datasets, contains the value the model will try to predict. However, this is only a convention, and some datasets will declare other sources, that can be used in varying ways by models, for instance when using multi-modal data.

A composite source is identified by a tuple of sources. For instance, to request features and targets from a dataset, the *source* would be `('features', 'targets')`.

7.5.5 Structure of data specifications

When using data specifications `data_specs=(space, source)`, `space` and `source` have to have the same *structure*. This means that:

- if `space` is an elementary space, then `source` has to be an elementary source, i.e., a string;
- if `space` is a composite space, then `source` has to be a composite source (a tuple), with exactly as many components as the number of sub-spaces of `space`; and the corresponding sub-sources and sub-spaces again have to have the same *structure*.

For example, let us define the following spaces:

```
input_vecspace = VectorSpace(dim=(32 * 32 * 3))
input_convspace = Conv2DSpace(shape=(32, 32), num_channels=3,
                              axes=('b', 'c', 0, 1))
target_space = VectorSpace(dim=10)
```

and suppose `"features"` and `"targets"` are sources present in our data. Then, the following data_specs are correct:

- `(input_vecspace, "features")`: only the features, mini-batches will be matrices;
- `(input_convspace, "features")`: only the features, mini-batches will be 4-D tensors;
- `(target_space, "targets")`: only the targets, mini-batches will be matrices;
- `(CompositeSpace((input_vecspace, target_space)), ("features", "targets"))`: features and targets, in that order; mini-batches will be (matrix, matrix) pairs;
- `(CompositeSpace((target_space, input_convspace)), ("targets", "features"))`: targets and features, in that order; mini-batches will be (matrix, 4-D tensor) pairs;

- `(CompositeSpace((input_vecspace, input_vecspace, input_vecspace, target_space)), ("features", "features", "features", "targets"))`: features repeated 3 times, then targets; mini-batches will be (matrix, matrix, matrix, matrix) tuples;
- `(CompositeSpace((CompositeSpace((input_vecspace, input_vecspace, input_vecspace)), target_space)), (("features", "features", "features"), "targets"))`: same as above, but the repeated features are in another CompositeSpace; mini-batches will be ((matrix, matrix, matrix), matrix) pairs with the first element being a triplet.

The following ones are **incorrect**:

- `(target_vecspace, "features")`: it will not crash immediately, but as soon as actual data are used, it will crash because feature data will have a width of 32 * 32 * 3 = 3072, but `target_vecspace.dim` is 10;
- `(CompositeSpace((input_vecspace, input_convspace)), "features")`: the `source` part has to have as many elements as there are sub-spaces of the `CompositeSpace`, but `"features"` is not a pair. You would need to write `(CompositeSpace((input_vecspace, input_convspace)), ("features", "features"))`;
- `(CompositeSpace((input_vecspace,)), "features")`: the `source` part should be a tuple of length 1, not a string. You would need to write `(CompositeSpace((input_vecspace,)), ("features",))`;
- `(CompositeSpace((input_vecspace, input_vecspace, input_vecspace, target_space)), (("features", "features", "features"), "targets"))`: even if the total number of elementary spaces and elementary sources match, their *structure* do not: the sub-spaces are in a flat tuple of length 4, the sources are in a nested tuple;
- `(CompositeSpace((CompositeSpace((input_vecspace, input_vecspace, input_vecspace)), target_space)), ("features", "features", "features", "targets"))`: it is the same problem, the other way around.

7.5.6 Examples of use

Here are some examples of how data specifications are currently used in different Pylearn2 objects.

The big picture

The `TrainingAlgorithm` object (for instance `DefaultTrainingAlgorithm`, or `SGD`) is usually the one requesting the data_specs from the various objects defined in an experiment script (model, costs, monitor channels), combines them in one nested data_specs, flattens it, requests iterators from the datasets, iterates over the dataset, converting back the flat version of the data so it can be correctly dispatched between all the objects requiring data.

Input of a model

A Model object used in an experiment has to declare its input source and space, so the right data will be provided to it by the dataset iterator, in the appropriate format. This is done by the methods `models.Model.get_input_source()` and `models.Model.get_input_space()`.

By default, most models will simply use `"features"` as input source, but that could be changed for an experiment where the user wants to apply the model on a different source of the dataset, or on a dataset where sources are named differently.

Models that do not care for the topology of the input will use a `VectorSpace` as input space, whereas convolutional models, for instance, will use an instance of `Conv2DSpace`.

Models also declare an output space, which can be useful for the cost, for instance, or for other objects that can use or embed a model.

Input of a cost

A Cost object needs to implement the `costs.Cost.get_data_specs(self, model)()` method, which will be used to determine which data (and format) will be passed as the `data` argument of `costs.Cost.expr(self, model, data)()` and `costs.Cost.get_gradients(self, model, data)()`.

Example 1: cost without data

For instance, a cost that does not depend on data at all, but only on the model parameters, like an L1 regularization penalty, would typically use `(NullSpace(), '')` for data specifications, and `expr` would be passed `data=None`.

Example 2: unsupervised cost

An unsupervised cost, that uses only unlabeled features, and not targets, will usually use `(model.get_input_space(), model.get_input_source())`, so the `data` passed to `expr` will directly be usable by the model.

Example 3: supervised cost

Finally, a supervised cost, needing both features and targets, will usually request the targets to be in the same space as the model's predictions (the model's output space):

```
def get_data_specs(self, model):
    return (CompositeSpace((model.get_input_space(),
                            model.get_output_space())),
            (model.get_input_source(),
             "targets"))
```

Then, `data` would be a pair, the first element of which can be passed directly to the model.

Of course, it does not have to be implemented that way, and the following is as correct (if more confusing) if you prefer having `data` be a (targets, inputs) pair instead:

```
def get_data_specs(self, model):
    return (CompositeSpace((model.get_output_space(),
                            model.get_input_space())),
            ("targets",
             model.get_input_source()))
```

Input of a monitoring channel

As for costs used for training, variables monitored by MonitorChannels have to declare data specs corresponding to the input variables necessary to compute the monitored value. It is passed directly to the constructor, for instance, when calling:

```
channel = MonitorChannel(
    graph_inputs=input_variables,
    val=monitored_value,
    name='channel_name',
    data_specs=data_specs,
    dataset=dataset)
```

`data_specs` describe the format and semantics of `input_variables`.

As in the previous section, if `val` does not need any input data, for instance if it is a shared variable, `data_specs` will be `(NullSpace(), '')`. If `val` corresponds to an unsupervised cost, or quantity depending only on the `"features"` source, `data_specs` could be `(VectorSpace(...), "features")`, etc.

For monitored values defined in `models.Model.get_monitoring_channels(self, data)()`, the data_specs of `data`, which are also the `data_specs` to pass to MonitorChannel's constructor, are returned by a call to `models.Model.get_monitoring_channels_data(self)()`.

Nesting and flattening data_specs

In order to avoid duplicating data and creating lots of symbolic inputs to Theano functions (which also do not support nested arguments), it can be useful to convert a nested, composite data_specs into a flat, non-redundant one. That *flat* data_specs can be used to create theano variables or get mini-batches of data, for instance, which are then *nested* back into the original *structure* of the data_specs.

We use the `utils.data_specs.DataSpecsMapping` class to build a *mapping* between the original, nested data specs, and the flat one. For instance, using the spaces defined earlier:

```
source = ("features", ("features", "targets"))
space = CompositeSpace((input_vecspace,
                        CompositeSpace((input_convspace,
                                        target_space))))
mapping = DataSpecsMapping((space, source))
flat_source = mapping.flatten(source)
```

```
# flat_source == ('features', 'features', 'targets')
flat_space = mapping.flatten(space)
# flat_space == (input_vecspace, input_convspace, target_space)

# We can use the mapping the other way around
nested_source = mapping.nest(flat_source)
assert source == flat_source
nested_space = mapping.nest(flat_space)
assert space == flat_space

# We can also nest other things
print mapping.nest((1, 2, 3))
# (1, (2, 3))
```

Here, `'features'` appear twice in the flat source, that is because the corresponding space is different. However, if there is an actual duplicate, it gets removed:

```
source = (("features", "targets"), ("features", "targets"))
space = CompositeSpace((CompositeSpace((input_vecspace, target_space)),
                        CompositeSpace((input_vecspace, target_space))))
mapping = DataSpecsMapping((space, source))
flat_source = mapping.flatten(source)
# flat_source == ('features', 'targets')
flat_space = mapping.flatten(space)
# flat_space == (input_vecspace, target_space)

# We can use the mapping the other way around
nested_source = mapping.nest(flat_source)
assert source == flat_source
nested_space = mapping.nest(flat_space)
assert space == flat_space

# We can also nest other things
print mapping.nest((1, 2))
# ((1, 2), (1, 2))
```

The flat tuple of spaces can be used to create non-redundant Theano input variables, which will be nested back to be dispatched between the different components having requested them:

```
# From the block above:
# flat_space == (input_vecspace, target_space)

flat_composite_space = CompositeSpace(flat_space)
flat_inputs = flat_composite_space.make_theano_variables(name='input')
print flat_inputs
# (input[0], input[1])

# We can use the mapping to nest the theano variables
nested_inputs = mapping.nest(theano_inputs)
print nested_inputs
# ((input[0], input[1]), (input[0], input[1]))

# Then, we can build expressions from these input variables.
# Finally, a Theano function will be compiled with
```

```
f = theano.function(flat_inputs, outputs, ...)

# A dataset iterator can also be created from the flat composite space
it = my_dataset.iterator(..., data_specs=(flat_composite_space, flat_source))

# When it is time to call f on data, we can then do
for flat_data in it:
    out = f(*flat_data)
```

7.5.7 Pylearn2 Pull Request Checklist

Last updated: June 8, 2014

This is a preliminary list of common pull request fixes requested. It's presumed that your pull request should already pass the Travis buildbot, including docstring and code formatting checks.

- *Are you breaking a statement over multiple lines?*
- *Do tests exist for the code you're modifying?*
- *Are you fixing a bug? Did you add a regression test?*
- *Are you fixing an issue that is on the issue tracker?*
- *Have you squashed out any nuisance commits?*
- *Are you using OrderedDict where necessary? Are you iterating over sets?*
- *Are you using print statements?*
- *Are you creating a sequence and then immediately iterating over it?*
- *Are you using* zip()/izip() *on sequences you expect to be the same length?*
- *Are you using the dict/*OrderedDict *methods* keys()/values()/items()*?*
- *Are you updating a dictionary or OrderedDict with* .update()*?*
- *Do you have an* except: *block?*
- *Are you checking to see if an argument is iterable?*
- *Are you checking if something is a string?*
- *Are you checking if something is a number?*
  - *Are you checking if something is _any_ kind of number?*
  - *Are you checking if something is an integer?*
  - *Are you checking if something is a float?*
  - *Are you checking if something is a complex number?*
  - *Are you checking for the presence of* np.nan *or* np.inf *in an array?*
- *Are you creating Theano functions?*
- *Are you creating Theano shared variables?*
- *Are you casting symbols/constants to a Theano floating point type?*
- *Do you have big nested loops for generating a Cartesian product?*
- *Are you generating combinations or permutations of a set (or list, ...)?*
- *Are you overriding methods in your class?*
- *Are you writing functions that uses pseudo-random numbers?*
- *Are you assembling filesystem paths with* dir + / + filename *or similar?*
- *Are you extracting the directory name or base filename from a file path?*
- *Are you opening/closing files?*

Are you breaking a statement over multiple lines?

Python supports breaking a logical line over multiple file lines in a number of ways. One is to use backslashes before the line ending. Another is to enclose the broken section is parentheses (), ([] also works but you should only use this if you are otherwise creating a list). Note that if you have open parentheses from a function call you do not need additional parentheses.

In Pylearn2 we generally prefer parentheses, because it means there's less markup to maintain and leads to less spurious errors.

Yes:

```
assert some_complicated_conditional_thing, (
    "This is the assertion error on a separate line."
)
```

No:

```
assert some_complicated_conditional_thing, \
    "This just gets annoying, especially if there are multiple " \
    "lines of text."
```

Note that string concatenation across lines is automatic, no need for +. If enclosed in parentheses you don't need a either:

```
# Valid Python.
print ("The quick brown fox jumps over the lazy dog. And then "
       "the fox did it again.")
```

See the PEP8 indentation recommendations for how to arrange indentation for continuation lines.

Do tests exist for the code you're modifying?

Pylearn2 grew rapidly in the beginning, often without proper attention to testing. Modifying a piece of code in the codebase may alter how it works; if you make such a modification, you should not only verify that tests pass but that tests _exist_ for the piece of code you're modifying. You should verify that those tests exist and update them as needed, including a test case for the behaviour you're adding or modifying.

Usually tests for a module *foo* are found in *tests/test_foo.py*.

Are you fixing a bug? Did you add a regression test?

Tests that test for previously existing bugs are particularly critical, as further modification of the code may reintroduce the bug by those who are not aware of the subtleties that led to it in the first place.

Are you fixing an issue that is on the issue tracker?

Your pull request *description* (or a commit message for one of the commits) should include one of the supported variants of the syntax so that the issue is auto-closed upon merge.

Have you squashed out any nuisance commits?

Pull requests with lots and lots of tiny commits are hard to review. Lots of commits that subsequently introduce a minor bug and then fix them can also make bisecting a pain.

Your final pull request should comprise as few commits as logically make sense. Each commit should ideally leave the repository in a working state (tests passing, functionality preserved).

You can squash commits using *git rebase -i* and following the instructions. Note that you will have to *git push –force origin my_branch_name* after a rebase.

You should squash to a minimal set of semantically distinct commits before asking for a review, and then possibly squash again if you've made lots of commits in response to feedback (note that you can reorder the commits in the editor window given by *git rebase -i*).

Are you using OrderedDict where necessary? Are you iterating over sets?

The order of iteration over dictionaries in Python is not guaranteed to remain the same across different invocations of the same program. This is a result of a randomized hashing algorithm and is actually an important security feature for preventing certain kinds of Denial-of-Service attacks. Unfortunately, where such data structures are employed in scientific simulations, this can pose reproducibility problems.

The main reason for this is that computations in floating point do not precisely obey the typical laws of arithmetic (commutativity, associativity, distributivity), and slight differences in the order of operations can introduce small differences in result, which can have butterfly effects that significantly alter the results of a long-running job. The order of operations can be altered by the order in which a Theano graph is assembled, and the precise form it takes can unfortunately sometimes alter which compile-time graph optimizations are performed.

In order to stamp out inconsistencies introduced by an unpredictable iteration order, we make extensive use of the *OrderedDict* class. This class is part of the *collections* module in Python 2.7 and Python 3.x, however, in order to maintain Python 2.6 compatibility, we import it from *theano.compat.python2x*, which provides an equivalent pure-Python implementation if the built-in version is not available.

You should consider carefully whether the iteration order over a dictionary you're using could result in different behaviour. If in doubt, use an *OrderedDict*. For the *updates* parameter when creating Theano functions, you _must_ use an *OrderedDict*, or a list of *(shared_variable, update_expression)* tuples.

When iterating over sets, consider whether you should first sort your set. The *sorted()* built-in function is a simple way of doing this.

Are you using print statements?

In most cases you should be using logging statements instead. You can initialize a logger in a new module with:

```
import logging

log = logging.getLogger(__name__)
```

And subsequently call into it with *log.info()*, *log.warning()*, etc.

Are you creating a sequence and then immediately iterating over it?

If so, consider using the faster and more memory efficient versions.

- *xrange* instead of *range*.
- *from theano.compat.six.moves import zip as izip* instead of *zip*. This import is for Python 3 compatibility.

Are you using *zip()*/*izip()* on sequences you expect to be the same length?

Note that zip and izip truncate the sequence of tuples they produce to the length of the shortest input sequence. If you expect, as is often the case, that the sequences you are zipping together should be the same length, use *safe_zip* or *safe_izip* defined in *pylearn2.utils*.

Also see *itertools.izip_longest* if you want to zip together sequences of unequal length with a fill value.

Are you using the dict/*OrderedDict* methods *keys()*/*values()*/*items()*?

For *values()* and *items()* consider whether *itervalues()* or *iteritems()* would be more appropriate, if you're only iterating over them once, not keeping the result around for any length of time, and don't need random access.

Also, don't bother with *keys()* or *iterkeys()* at all if you're just going to iterate over it. *for k in my_dictionary* iterates over keys by default.

An exception to these rules is if you are _modifying_ the dictionary within the loop. Then you probably want to duplicate things with the *keys()*, *values()* and *items()* calls.

Are you updating a dictionary or OrderedDict with *.update()*?

If you are using the *update()* method of a dictionary or *OrderedDict* and you expect that none of the keys in the argument should already be in the dictionary, use *safe_update()* defined in *pylearn2.utils*.

Do you have an *except:* block?

You should almost never have a bare *except:* in library code. Use:

```
except Exception:
    ...
```

instead. This catches any subclass of *Exception* but lets through certain low-level exceptions like *KeyboardInterrupt*, *SystemExit*, etc. that inherit from *BaseException* instead. You almost certainly do not want your code to catch these.

Don't raise a new exception, use the *reraise_as* method from *pylearn2.utils.exc* instead.:

```
except Exception:
    reraise_as(ValueError("Informative error message here"))
```

This retains the traceback and original error message, allowing for easier debugging using a tool like pdb.

Are you checking to see if an argument is iterable?

In places where a list, tuple, or other iterable object (say, a deque) will suffice, use *pylearn2.utils.is_iterable*.

Are you checking if something is a string?

Unless you have a very good reason you should probably be using *isinstance(foo, basestring)* which correctly handles both *str* and *unicode* instances.

Are you checking if something is a number?

Usually such checks are unnecessary but where they might be, we've defined some helpful constants.

Are you checking if something is _any_ kind of number?

Use *isinstance(foo, pylearn2.utils.py_number_types)*. This checks against Python builtins as well as NumPy-defined numerical types.

Are you checking if something is an integer?

Use *isinstance(foo, pylearn2.utils.py_integer_types)*. This checks against Python builtins as well as NumPy-defined integer types.

Are you checking if something is a float?

First, ask yourself: do you really need to? Would passing an integer here be inappropriate in all circumstances? Would a cast (i.e. *float()* be sufficient?

If you really need to, use *isinstance(foo, pylearn2.utils.py_float_types)*. This checks against Python builtins as well as NumPy-defined float types.

Are you checking if something is a complex number?

Again, ask yourself whether passing a real here would be an error, and whether you can get away with a cast.

If you really need to, use *isinstance(foo, pylearn2.utils.py_complex_types)*. This checks against Python builtins as well as NumPy-defined complex types.

Are you checking for the presence of *np.nan* or *np.inf* in an array?

If so, use *pylearn2.utils.contains_nan* or *pylearn2.utils.contains_inf*. To check for either *np.nan* or *np.inf*, use *pylearn2.utils.isfinite*. These functions are faster and more memory effecient than *np.any(np.isnan(X))* or *np.any(np.isinf(X))*.

Are you creating Theano functions?

If you're building Theano functions, use *pylearn2.utils.function*. This disables the *on_unused_input* check, which in most cases you don't want to consider an error if you're doing any kind of generic graph building.

Are you creating Theano shared variables?

If your model, cost, etc. creates floating point shared variables you should probably create them with *pylearn2.utils.sharedX*, which creates shared variables with the dtype set to *theano.config.floatX*.

Are you casting symbols/constants to a Theano floating point type?

Use *pylearn2.utils.as_floatX* to cast symbolic quantities to the default floating point type, and use *constantX* to create symbolic constants from a scalar or ndarray with the dtype specified in *theano.config.floatX*.

Do you have big nested loops for generating a Cartesian product?

Example:

```
stuff = []
for i in range(50):
    for j in range(20):
        for k in range(30):
            stuff.append((i, j, k))
```

Consider whether *itertools.product* will get the job done more readably and probably more efficiently.

Are you generating combinations or permutations of a set (or list, ...)?

itertools contains the functions *permutations*, *combinations* and *combinations_with_replacement* that will probably get the job done more efficiently than your own code.

Are you overriding methods in your class?

Use the decorator *pylearn2.utils.wraps* to inherit the docstring if it is unchanged. If you add a docstring to a function that is wrapped in this fashion, it will be appended below the inherited docstring.

Are you writing functions that uses pseudo-random numbers?

If you are using the NumPy generator, are you providing a way to seed it as well as a default seed ? You should **never** be using *numpy.random* functions directly. Use *pylearn2.utils.rng.make_np_rng* with a user-provided seed and a *default_seed* argument.

If you are using the Theano RNG you should create it similarly with *pylearn2.utils.rng.make_theano_rng*.

Are you assembling filesystem paths with *dir* + / + *filename* or similar?

Use *os.path.join* rather than concatenating together with '/'. This ensures the code still works on Windows.

Are you extracting the directory name or base filename from a file path?

Use *os.path.basename* and *os.path.dirname* to ensure Windows compatibility.

Are you opening/closing files?

Use the *with* statement, i.e.:

```
with open(fname, 'w') as f:
    f.write('blah blah blah')
```

This is cleaner and ensures that the file always gets closed, even in an error condition.

7.6 Documentation Documentation AKA Meta-Documentation

7.6.1 How to build documentation

Let's say you are writing documentation, and want to see the sphinx output before you push it. The documentation will be generated in the `html` directory.

```
cd Pylearn2/
python ./doc/scripts/docgen.py
```

If you don't want to generate the pdf, do the following:

```
cd Pylearn2/
python ./doc/scripts/docgen.py --nopdf
```

For more details:

```
$ python doc/scripts/docgen.py --help
Usage: doc/scripts/docgen.py [OPTIONS]
  -o <dir>: output the html files in the specified dir
  --rst: only compile the doc (requires sphinx)
  --nopdf: do not produce a PDF file from the doc, only HTML
  --help: this help
```

7.6.2 Use ReST for documentation

- ReST is standardized. epydoc is not. trac wiki-markup is not. This means that ReST can be cut-and-pasted between epydoc, code, other docs, and TRAC. This is a huge win!
- ReST is extensible: we can write our own roles and directives to automatically link to WIKI, for example.
- ReST has figure and table directives, and can be converted (using a standard tool) to latex documents.
- No text documentation has good support for math rendering, but ReST is closest: it has three renderer-specific solutions (render latex, use latex to build images for html, use itex2mml to generate MathML)

7.6.3 How to link to class/function documentations

Link to the generated doc of a function this way:

```
:func:`perform`
```

For example:

```
of the :func:`perform` function.
```

Link to the generated doc of a class this way:

```
:class:`RopLop_checker`
```

For example:

```
The class :class:`RopLop_checker`, give the functions
```

However, if the link target is ambiguous, Sphinx will generate warning or errors.

7.6.4 How to add TODO comments in Sphinx documentation

To include a TODO comment in Sphinx documentation, use an indented block as follows:

```
.. TODO: This is a comment.
.. You have to put .. at the beginning of every line :(
.. These lines should all be indented.
```

It will not appear in the output generated.

7.6.5 How documentation is built on deeplearning.net

The server that hosts the pylearn2 documentation runs a cron job roughly every 2 hours that fetches a fresh Pylearn2 install (clone, not just pull) and executes the docgen.py script. It then over-writes the previous docs with the newly generated ones.

Note that the server will most definitely use a different version of sphinx than yours so formatting could be slightly off, or even wrong. If you're getting unxpected results and/or the auto-build of the documentation seems broken, please contact pylearn-dev@.

In the future, we might go back to the system of auto-refresh on push (though that might increase the load of the server quite significantly).

7.6.6 The nightly build/tests process

The user `lisa` runs a cronjob on the computer `ceylon`, this happens nightly. (To have the crontab executed, the `lisa` user must be logged into `ceylon`, Fred leaves a shell open for that.)

The cronjob executes the scripts `~/nightly_build/do_nightly_build_{theano,pylearn,deeplearning` These scripts perform an update of theano (and pylearn, and DeepLearningTutorials too), and executes nose (in various settings).

The output is emailed automatically to the theano-buildbot mailing list.

7.6.7 TO WRITE

There is other stuff to document here, e.g.:

- We also want examples of good documentation, to show people how to write ReST.

7.7 Data specifications, spaces, and sources

Data specifications, often called `data_specs`, are used as a specification for requesting and providing data in a certain format across different parts of a Pylearn2 experiment.

A `data_specs` is a `(space, source)` pair, where `source` is an identifier or the data source or sources required (for instance, inputs and targets), and `space` is an instance of `space.Space` representing the *format* of these data (for instance, a vector, a 3D tensor representing an RGB image, or a one-hot vector).

The main use of `data_specs` is to request data from a `datasets.Dataset` object, via an iterator. Various objects can request data this way: models, costs, monitoring channels, training algorithms, even some datasets that perform a transformation on data.

7.8 The `Space` object

A `Space` represents a way in which a mini-batch of data can be formatted. For instance, a batch of RGB images (each of shape `(rows, columns)`) can be represented in different ways, for instance:

- as a matrix where each row corresponds to a different image, and is of length `rows * columns * 3`: the corresponding space would be a `space.VectorSpace`, more precisely `VectorSpace(dim=(rows * columns * 3))`;

- as a 4-dimensional tensor, where rows, columns, and channels (here: red, green, and blue) are different axes: the corresponding space would be a `space.Conv2DSpace`. Theano convolutions prefer that tensor to have shape `(batch_size, channels, rows, columns)`, which corresponds to `Conv2DSpace(shape=(rows, columns), num_channels=3, axes=('b', 'c', 0, 1))`;
- as a 4-dimensional tensors with a different shape: for instance, cuda-convnet prefers `(channels, rows, columns, batch_size)`: the space would be `Conv2DSpace(shape=(rows, columns), num_channels=3, axes=('c', 0, 1, 'b'))`.

Spaces can be either elementary, representing one mini-batch from one source of data, such as `VectorSpace` and `Conv2DSpace` mentioned above, or *composite* (`space.CompositeSpace`), representing the aggregation of several sources of data (some of these may in turn be aggregations of sources). A mini-batch for an elementary space will usually be a NumPy `ndarray`, whereas a mini-batch for a `CompositeSpace` will be a Python tuple of elementary (or composite) mini-batches.

Notable methods of the `Space` class are:

- `Space.`**`make_theano_batch`**`()`

  creates a Theano Variable (or tuple of Theano Variable in the case of `CompositeSpace`) representing a *symbolic* mini-batch of data. For instance, `VectorSpace(...).make_theano_batch(...)` will essentially call `theano.tensor.matrix()`.

- `Space.`**`validate`**`(`*batch*`)`

  will check that symbolic variable `batch` can correctly represent a mini-batch of data for the corresponding space. For instance, `VectorSpace(...).validate(theano.tensor.matrix())` will work, but `VectorSpace(...).validate(theano.tensor.vector())` will raise an exception.

- `Space.`**`np_validate`**`(`*batch*`)`

  (`np` stands for NumPy) is similar, but operates on a mini-batch of numeric data, rather than on a symbolic variable. This enables more checks to be performed. For instance, `VectorSpace(dim=3).validate(np.zeros((4, 3)))` will work, because it correctly describes a mini-batch of 4 samples of dimension 3, but `VectorSpace(dim=4).validate(np.zeros((4, 3)))` will raise an exception.

- `Space.`**`format_as`**`(`*batch*, *space*`)`

  and

  `Space.`**`np_format_as`**`(`*batch*, *space*`)`

  are the way we can convert data from their original space into the destination `space`. `format_as` operates on a symbolic `batch`, and returns a symbolic expression of the newly-formatted data, whereas `np_format_as` operates on a numeric batch, and returns numeric data. This formatting can happen between different instances of the same `Space` class, for instance, converting between two instances of `Conv2DSpace` with different `axes` amounts to correctly transpose the `batch`. It can also happen between different subclasses of `Space`, for instance, converting between a `VectorSpace` and `Conv2DSpace` of compatible shape involves reshaping and transposition of the data.

7.9 Sources

Sources are simple identifiers that specify *which* data should be returned, whereas spaces specify *how* that data should be formatted.

An elementary source is identified by a Python string. For instance, the most used sources are `'features'` and `'targets'`. `'features'` usually denotes the part of the data that models will use as input, and `'targets'`, for labeled datasets, contains the value the model will try to predict. However, this is only a convention, and some datasets will declare other sources, that can be used in varying ways by models, for instance when using multi-modal data.

A composite source is identified by a tuple of sources. For instance, to request features and targets from a dataset, the *source* would be `('features', 'targets')`.

7.10 Structure of data specifications

When using data specifications `data_specs=(space, source)`, `space` and `source` have to have the same *structure*. This means that:

- if `space` is an elementary space, then `source` has to be an elementary source, i.e., a string;
- if `space` is a composite space, then `source` has to be a composite source (a tuple), with exactly as many components as the number of sub-spaces of `space`; and the corresponding sub-sources and sub-spaces again have to have the same *structure*.

For example, let us define the following spaces:

```
input_vecspace = VectorSpace(dim=(32 * 32 * 3))
input_convspace = Conv2DSpace(shape=(32, 32), num_channels=3,
                              axes=('b', 'c', 0, 1))
target_space = VectorSpace(dim=10)
```

and suppose `"features"` and `"targets"` are sources present in our data. Then, the following data_specs are correct:

- `(input_vecspace, "features")`: only the features, mini-batches will be matrices;
- `(input_convspace, "features")`: only the features, mini-batches will be 4-D tensors;
- `(target_space, "targets")`: only the targets, mini-batches will be matrices;
- `(CompositeSpace((input_vecspace, target_space)), ("features", "targets"))`: features and targets, in that order; mini-batches will be (matrix, matrix) pairs;
- `(CompositeSpace((target_space, input_convspace)), ("targets", "features"))`: targets and features, in that order; mini-batches will be (matrix, 4-D tensor) pairs;
- `(CompositeSpace((input_vecspace, input_vecspace, input_vecspace, target_space)), ("features", "features", "features", "targets"))`: features repeated 3 times, then targets; mini-batches will be (matrix, matrix, matrix, matrix) tuples;

- `(CompositeSpace((CompositeSpace((input_vecspace, input_vecspace, input_vecspace)), target_space)), (("features", "features", "features"), "targets"))`: same as above, but the repeated features are in another CompositeSpace; mini-batches will be ((matrix, matrix, matrix), matrix) pairs with the first element being a triplet.

The following ones are **incorrect**:

- `(target_vecspace, "features")`: it will not crash immediately, but as soon as actual data are used, it will crash because feature data will have a width of 32 * 32 * 3 = 3072, but `target_vecspace.dim` is 10;
- `(CompositeSpace((input_vecspace, input_convspace)), "features")`: the `source` part has to have as many elements as there are sub-spaces of the `CompositeSpace`, but `"features"` is not a pair. You would need to write `(CompositeSpace((input_vecspace, input_convspace)), ("features", "features"))`;
- `(CompositeSpace((input_vecspace,)), "features")`: the `source` part should be a tuple of length 1, not a string. You would need to write `(CompositeSpace((input_vecspace,)), ("features",))`;
- `(CompositeSpace((input_vecspace, input_vecspace, input_vecspace, target_space)), (("features", "features", "features"), "targets"))`: even if the total number of elementary spaces and elementary sources match, their *structure* do not: the sub-spaces are in a flat tuple of length 4, the sources are in a nested tuple;
- `(CompositeSpace((CompositeSpace((input_vecspace, input_vecspace, input_vecspace)), target_space)), ("features", "features", "features", "targets"))`: it is the same problem, the other way around.

7.11 Examples of use

Here are some examples of how data specifications are currently used in different Pylearn2 objects.

7.11.1 The big picture

The `TrainingAlgorithm` object (for instance `DefaultTrainingAlgorithm`, or `SGD`) is usually the one requesting the data_specs from the various objects defined in an experiment script (model, costs, monitor channels), combines them in one nested data_specs, flattens it, requests iterators from the datasets, iterates over the dataset, converting back the flat version of the data so it can be correctly dispatched between all the objects requiring data.

7.11.2 Input of a model

A Model object used in an experiment has to declare its input source and space, so the right data will be provided to it by the dataset iterator, in the appropriate format. This is done by the methods `models.Model.get_input_source()` and `models.Model.get_input_space()`.

By default, most models will simply use `"features"` as input source, but that could be changed for an experiment where the user wants to apply the model on a different source of the dataset, or on a dataset where sources are named differently.

Models that do not care for the topology of the input will use a `VectorSpace` as input space, whereas convolutional models, for instance, will use an instance of `Conv2DSpace`.

Models also declare an output space, which can be useful for the cost, for instance, or for other objects that can use or embed a model.

7.11.3 Input of a cost

A Cost object needs to implement the `costs.Cost.get_data_specs(self, model)()` method, which will be used to determine which data (and format) will be passed as the `data` argument of `costs.Cost.expr(self, model, data)()` and `costs.Cost.get_gradients(self, model, data)()`.

Example 1: cost without data

For instance, a cost that does not depend on data at all, but only on the model parameters, like an L1 regularization penalty, would typically use `(NullSpace(), '')` for data specifications, and `expr` would be passed `data=None`.

Example 2: unsupervised cost

An unsupervised cost, that uses only unlabeled features, and not targets, will usually use `(model.get_input_space(), model.get_input_source())`, so the `data` passed to `expr` will directly be usable by the model.

Example 3: supervised cost

Finally, a supervised cost, needing both features and targets, will usually request the targets to be in the same space as the model's predictions (the model's output space):

```
def get_data_specs(self, model):
    return (CompositeSpace((model.get_input_space(),
                            model.get_output_space())),
            (model.get_input_source(),
             "targets"))
```

Then, `data` would be a pair, the first element of which can be passed directly to the model.

Of course, it does not have to be implemented that way, and the following is as correct (if more confusing) if you prefer having `data` be a (targets, inputs) pair instead:

```
def get_data_specs(self, model):
    return (CompositeSpace((model.get_output_space(),
                            model.get_input_space())),
```

```
            ("targets",
             model.get_input_source()))
```

7.11.4 Input of a monitoring channel

As for costs used for training, variables monitored by MonitorChannels have to declare data specs corresponding to the input variables necessary to compute the monitored value. It is passed directly to the constructor, for instance, when calling:

```
channel = MonitorChannel(
    graph_inputs=input_variables,
    val=monitored_value,
    name='channel_name',
    data_specs=data_specs,
    dataset=dataset)
```

`data_specs` describe the format and semantics of `input_variables`.

As in the previous section, if `val` does not need any input data, for instance if it is a shared variable, `data_specs` will be `(NullSpace(), '')`. If `val` corresponds to an unsupervised cost, or quantity depending only on the `"features"` source, `data_specs` could be `(VectorSpace(...), "features")`, etc.

For monitored values defined in `models.Model.get_monitoring_channels(self, data)()`, the data_specs of `data`, which are also the `data_specs` to pass to MonitorChannel's constructor, are returned by a call to `models.Model.get_monitoring_channels_data(self)()`.

7.11.5 Nesting and flattening data_specs

In order to avoid duplicating data and creating lots of symbolic inputs to Theano functions (which also do not support nested arguments), it can be useful to convert a nested, composite data_specs into a flat, non-redundant one. That *flat* data_specs can be used to create theano variables or get mini-batches of data, for instance, which are then *nested* back into the original *structure* of the data_specs.

We use the `utils.data_specs.DataSpecsMapping` class to build a *mapping* between the original, nested data specs, and the flat one. For instance, using the spaces defined earlier:

```
source = ("features", ("features", "targets"))
space = CompositeSpace((input_vecspace,
                        CompositeSpace((input_convspace,
                                        target_space))))
mapping = DataSpecsMapping((space, source))
flat_source = mapping.flatten(source)
# flat_source == ('features', 'features', 'targets')
flat_space = mapping.flatten(space)
# flat_space == (input_vecspace, input_convspace, target_space)

# We can use the mapping the other way around
nested_source = mapping.nest(flat_source)
assert source == flat_source
```

```
nested_space = mapping.nest(flat_space)
assert space == flat_space

# We can also nest other things
print mapping.nest((1, 2, 3))
# (1, (2, 3))
```

Here, `'features'` appear twice in the flat source, that is because the corresponding space is different. However, if there is an actual duplicate, it gets removed:

```
source = (("features", "targets"), ("features", "targets"))
space = CompositeSpace((CompositeSpace((input_vecspace, target_space)),
                        CompositeSpace((input_vecspace, target_space))))
mapping = DataSpecsMapping((space, source))
flat_source = mapping.flatten(source)
# flat_source == ('features', 'targets')
flat_space = mapping.flatten(space)
# flat_space == (input_vecspace, target_space)

# We can use the mapping the other way around
nested_source = mapping.nest(flat_source)
assert source == flat_source
nested_space = mapping.nest(flat_space)
assert space == flat_space

# We can also nest other things
print mapping.nest((1, 2))
# ((1, 2), (1, 2))
```

The flat tuple of spaces can be used to create non-redundant Theano input variables, which will be nested back to be dispatched between the different components having requested them:

```
# From the block above:
# flat_space == (input_vecspace, target_space)

flat_composite_space = CompositeSpace(flat_space)
flat_inputs = flat_composite_space.make_theano_variables(name='input')
print flat_inputs
# (input[0], input[1])

# We can use the mapping to nest the theano variables
nested_inputs = mapping.nest(theano_inputs)
print nested_inputs
# ((input[0], input[1]), (input[0], input[1]))

# Then, we can build expressions from these input variables.
# Finally, a Theano function will be compiled with
f = theano.function(flat_inputs, outputs, ...)

# A dataset iterator can also be created from the flat composite space
it = my_dataset.iterator(..., data_specs=(flat_composite_space, flat_source))

# When it is time to call f on data, we can then do
for flat_data in it:
```

```
    out = f(*flat_data)
```

7.12 Pylearn2 Pull Request Checklist

Last updated: June 8, 2014

This is a preliminary list of common pull request fixes requested. It's presumed that your pull request should already pass the Travis buildbot, including docstring and code formatting checks.

- *Are you breaking a statement over multiple lines?*
- *Do tests exist for the code you're modifying?*
- *Are you fixing a bug? Did you add a regression test?*
- *Are you fixing an issue that is on the issue tracker?*
- *Have you squashed out any nuisance commits?*
- *Are you using OrderedDict where necessary? Are you iterating over sets?*
- *Are you using print statements?*
- *Are you creating a sequence and then immediately iterating over it?*
- *Are you using* zip()/izip() *on sequences you expect to be the same length?*
- *Are you using the dict/*OrderedDict *methods* keys()/values()/items()*?*
- *Are you updating a dictionary or OrderedDict with* .update()*?*
- *Do you have an* except: *block?*
- *Are you checking to see if an argument is iterable?*
- *Are you checking if something is a string?*
- *Are you checking if something is a number?*
  - *Are you checking if something is _any_ kind of number?*
  - *Are you checking if something is an integer?*
  - *Are you checking if something is a float?*
  - *Are you checking if something is a complex number?*
  - *Are you checking for the presence of* np.nan *or* np.inf *in an array?*
- *Are you creating Theano functions?*
- *Are you creating Theano shared variables?*
- *Are you casting symbols/constants to a Theano floating point type?*
- *Do you have big nested loops for generating a Cartesian product?*
- *Are you generating combinations or permutations of a set (or list, ...)?*
- *Are you overriding methods in your class?*
- *Are you writing functions that uses pseudo-random numbers?*
- *Are you assembling filesystem paths with* dir + / + filename *or similar?*
- *Are you extracting the directory name or base filename from a file path?*
- *Are you opening/closing files?*

7.12.1 Are you breaking a statement over multiple lines?

Python supports breaking a logical line over multiple file lines in a number of ways. One is to use backslashes before the line ending. Another is to enclose the broken section is parentheses (), ([] also works but

you should only use this if you are otherwise creating a list). Note that if you have open parentheses from a function call you do not need additional parentheses.

In Pylearn2 we generally prefer parentheses, because it means there's less markup to maintain and leads to less spurious errors.

Yes:

```
assert some_complicated_conditional_thing, (
    "This is the assertion error on a separate line."
)
```

No:

```
assert some_complicated_conditional_thing, \
    "This just gets annoying, especially if there are multiple " \
    "lines of text."
```

Note that string concatenation across lines is automatic, no need for +. If enclosed in parentheses you don't need a either:

```
# Valid Python.
print ("The quick brown fox jumps over the lazy dog. And then "
       "the fox did it again.")
```

See the PEP8 indentation recommendations for how to arrange indentation for continuation lines.

7.12.2 Do tests exist for the code you're modifying?

Pylearn2 grew rapidly in the beginning, often without proper attention to testing. Modifying a piece of code in the codebase may alter how it works; if you make such a modification, you should not only verify that tests pass but that tests _exist_ for the piece of code you're modifying. You should verify that those tests exist and update them as needed, including a test case for the behaviour you're adding or modifying.

Usually tests for a module *foo* are found in *tests/test_foo.py*.

7.12.3 Are you fixing a bug? Did you add a regression test?

Tests that test for previously existing bugs are particularly critical, as further modification of the code may reintroduce the bug by those who are not aware of the subtleties that led to it in the first place.

7.12.4 Are you fixing an issue that is on the issue tracker?

Your pull request *description* (or a commit message for one of the commits) should include one of the supported variants of the syntax so that the issue is auto-closed upon merge.

7.12.5 Have you squashed out any nuisance commits?

Pull requests with lots and lots of tiny commits are hard to review. Lots of commits that subsequently introduce a minor bug and then fix them can also make bisecting a pain.

Your final pull request should comprise as few commits as logically make sense. Each commit should ideally leave the repository in a working state (tests passing, functionality preserved).

You can squash commits using *git rebase -i* and following the instructions. Note that you will have to *git push –force origin my_branch_name* after a rebase.

You should squash to a minimal set of semantically distinct commits before asking for a review, and then possibly squash again if you've made lots of commits in response to feedback (note that you can reorder the commits in the editor window given by *git rebase -i*).

7.12.6 Are you using OrderedDict where necessary? Are you iterating over sets?

The order of iteration over dictionaries in Python is not guaranteed to remain the same across different invocations of the same program. This is a result of a randomized hashing algorithm and is actually an important security feature for preventing certain kinds of Denial-of-Service attacks. Unfortunately, where such data structures are employed in scientific simulations, this can pose reproducibility problems.

The main reason for this is that computations in floating point do not precisely obey the typical laws of arithmetic (commutativity, associativity, distributivity), and slight differences in the order of operations can introduce small differences in result, which can have butterfly effects that significantly alter the results of a long-running job. The order of operations can be altered by the order in which a Theano graph is assembled, and the precise form it takes can unfortunately sometimes alter which compile-time graph optimizations are performed.

In order to stamp out inconsistencies introduced by an unpredictable iteration order, we make extensive use of the *OrderedDict* class. This class is part of the *collections* module in Python 2.7 and Python 3.x, however, in order to maintain Python 2.6 compatibility, we import it from *theano.compat.python2x*, which provides an equivalent pure-Python implementation if the built-in version is not available.

You should consider carefully whether the iteration order over a dictionary you're using could result in different behaviour. If in doubt, use an *OrderedDict*. For the *updates* parameter when creating Theano functions, you _must_ use an *OrderedDict*, or a list of *(shared_variable, update_expression)* tuples.

When iterating over sets, consider whether you should first sort your set. The *sorted()* built-in function is a simple way of doing this.

7.12.7 Are you using print statements?

In most cases you should be using logging statements instead. You can initialize a logger in a new module with:

```
import logging

log = logging.getLogger(__name__)
```

And subsequently call into it with *log.info()*, *log.warning()*, etc.

7.12.8 Are you creating a sequence and then immediately iterating over it?

If so, consider using the faster and more memory efficient versions.

- *xrange* instead of *range*.
- *from theano.compat.six.moves import zip as izip* instead of *zip*. This import is for Python 3 compatibility.

7.12.9 Are you using *zip()*/*izip()* on sequences you expect to be the same length?

Note that zip and izip truncate the sequence of tuples they produce to the length of the shortest input sequence. If you expect, as is often the case, that the sequences you are zipping together should be the same length, use *safe_zip* or *safe_izip* defined in *pylearn2.utils*.

Also see *itertools.izip_longest* if you want to zip together sequences of unequal length with a fill value.

7.12.10 Are you using the dict/*OrderedDict* methods *keys()*/*values()*/*items()*?

For *values()* and *items()* consider whether *itervalues()* or *iteritems()* would be more appropriate, if you're only iterating over them once, not keeping the result around for any length of time, and don't need random access.

Also, don't bother with *keys()* or *iterkeys()* at all if you're just going to iterate over it. *for k in my_dictionary* iterates over keys by default.

An exception to these rules is if you are _modifying_ the dictionary within the loop. Then you probably want to duplicate things with the *keys()*, *values()* and *items()* calls.

7.12.11 Are you updating a dictionary or OrderedDict with *.update()*?

If you are using the *update()* method of a dictionary or *OrderedDict* and you expect that none of the keys in the argument should already be in the dictionary, use *safe_update()* defined in *pylearn2.utils*.

7.12.12 Do you have an *except:* block?

You should almost never have a bare *except:* in library code. Use:

```
except Exception:
    ...
```

instead. This catches any subclass of *Exception* but lets through certain low-level exceptions like *KeyboardInterrupt*, *SystemExit*, etc. that inherit from *BaseException* instead. You almost certainly do not want your code to catch these.

Don't raise a new exception, use the *reraise_as* method from *pylearn2.utils.exc* instead.:

```
except Exception:
    reraise_as(ValueError("Informative error message here"))
```

This retains the traceback and original error message, allowing for easier debugging using a tool like pdb.

7.12.13 Are you checking to see if an argument is iterable?

In places where a list, tuple, or other iterable object (say, a deque) will suffice, use *pylearn2.utils.is_iterable*.

7.12.14 Are you checking if something is a string?

Unless you have a very good reason you should probably be using *isinstance(foo, basestring)* which correctly handles both *str* and *unicode* instances.

7.12.15 Are you checking if something is a number?

Usually such checks are unnecessary but where they might be, we've defined some helpful constants.

Are you checking if something is _any_ kind of number?

Use *isinstance(foo, pylearn2.utils.py_number_types)*. This checks against Python builtins as well as NumPy-defined numerical types.

Are you checking if something is an integer?

Use *isinstance(foo, pylearn2.utils.py_integer_types)*. This checks against Python builtins as well as NumPy-defined integer types.

Are you checking if something is a float?

First, ask yourself: do you really need to? Would passing an integer here be inappropriate in all circumstances? Would a cast (i.e. *float()* be sufficient?

If you really need to, use *isinstance(foo, pylearn2.utils.py_float_types)*. This checks against Python builtins as well as NumPy-defined float types.

Are you checking if something is a complex number?

Again, ask yourself whether passing a real here would be an error, and whether you can get away with a cast.

If you really need to, use *isinstance(foo, pylearn2.utils.py_complex_types)*. This checks against Python builtins as well as NumPy-defined complex types.

Are you checking for the presence of *np.nan* or *np.inf* in an array?

If so, use *pylearn2.utils.contains_nan* or *pylearn2.utils.contains_inf*. To check for either *np.nan* or *np.inf*, use *pylearn2.utils.isfinite*. These functions are faster and more memory effecient than *np.any(np.isnan(X))* or *np.any(np.isinf(X))*.

7.12.16 Are you creating Theano functions?

If you're building Theano functions, use *pylearn2.utils.function*. This disables the *on_unused_input* check, which in most cases you don't want to consider an error if you're doing any kind of generic graph building.

7.12.17 Are you creating Theano shared variables?

If your model, cost, etc. creates floating point shared variables you should probably create them with *pylearn2.utils.sharedX*, which creates shared variables with the dtype set to *theano.config.floatX*.

7.12.18 Are you casting symbols/constants to a Theano floating point type?

Use *pylearn2.utils.as_floatX* to cast symbolic quantities to the default floating point type, and use *constantX* to create symbolic constants from a scalar or ndarray with the dtype specified in *theano.config.floatX*.

7.12.19 Do you have big nested loops for generating a Cartesian product?

Example:

```
stuff = []
for i in range(50):
    for j in range(20):
        for k in range(30):
            stuff.append((i, j, k))
```

Consider whether *itertools.product* will get the job done more readably and probably more efficiently.

7.12.20 Are you generating combinations or permutations of a set (or list, ...)?

itertools contains the functions *permutations*, *combinations* and *combinations_with_replacement* that will probably get the job done more efficiently than your own code.

7.12.21 Are you overriding methods in your class?

Use the decorator *pylearn2.utils.wraps* to inherit the docstring if it is unchanged. If you add a docstring to a function that is wrapped in this fashion, it will be appended below the inherited docstring.

7.12.22 Are you writing functions that uses pseudo-random numbers?

If you are using the NumPy generator, are you providing a way to seed it as well as a default seed ? You should **never** be using *numpy.random* functions directly. Use *pylearn2.utils.rng.make_np_rng* with a user-provided seed and a *default_seed* argument.

If you are using the Theano RNG you should create it similarly with *pylearn2.utils.rng.make_theano_rng*.

7.12.23 Are you assembling filesystem paths with *dir* + / + *filename* or similar?

Use *os.path.join* rather than concatenating together with '/'. This ensures the code still works on Windows.

7.12.24 Are you extracting the directory name or base filename from a file path?

Use *os.path.basename* and *os.path.dirname* to ensure Windows compatibility.

7.12.25 Are you opening/closing files?

Use the *with* statement, i.e.:

```
with open(fname, 'w') as f:
    f.write('blah blah blah')
```

This is cleaner and ensures that the file always gets closed, even in an error condition.

7.13 Coding Style Guidelines

7.13.1 Main Goals

- Code should be compatible with Python 2.4 and above (using 2to3 for conversion to Python 3.x). This may not be possible in the short term for Theano-dependent code.
- Code should be easy to read, understand and update by developers and users.
- Code should be well-documented and well-tested.

7.13.2 Python Coding Guidelines

Official Guidelines

Source Material

The four main documents describing our Python coding guidelines are:

- PEP 8 – Style Guide for Python Code
- Google Python Style Guide
- PEP 257 – Docstring Conventions
- Numpy Docstring Standard

However, there are a few points mentioned in those documents that we decided to do differently:

- Use only one space (not two) after a sentence-ending period in comments.

```
# Good.
# This is the first sentence. It is followed by a single blank space.
# Bad.
# This is the first sentence.  It is followed by two blank spaces.
```

- You do not need to add an extra blank line before the closing quotes of a multi-line docstring. Also, we ask that the first line of a multi-line docstring should contain only the opening quotes.

```
# Good.
"""
This is a multi-line docstring.

Which means it has more than one line.
"""

# Bad.
"""This is a multi-line docstring.

Which means it has more than one line.

"""
```

- Standard library imports can (and should) be on the same line, to avoid wasting space on straighforward imports:

```
# Good.
import os, sys, time
# Good when it does not fit on a single line.
import std_lib_module_1, std_lib_module_2, std_lib_module_3
import std_lib_module_4, std_lib_module_5, std_lib_module_6
# Bad.
import os
import sys
import time
```

- Importing class / functions from a module is allowed when these are used multiple times, and no ambiguity is possible.

```
# Good when Bar and Blah are used many times.
from foo import Bar, Blah
do_something_with(Bar(), Blah(), Bar(), Blah(), Bar(), Blah())
# Good in most situations.
import foo
do_something_with(foo.Bar(), foo.Blah())
# Bad.
from foo import *
from numpy import any    # Potential ambiguity with __builtin__.any
```

Excerpts

We emphasize here a few important topics that are found in the official guidelines:

- Only use ASCII characters in code files.

- Code indent must be done with four blank characters (no tabs).
- Limit lines to 79 characters.
- No trailing spaces.
- Naming conventions: `ClassName`, `TOP_LEVEL_CONSTANT`, `everything_else`.
- Comments should start with a capital letter (unless the first word is a code identifier) and end with a period (short inline comments may skip the period at the end).
- Imports should be listed in alphabetical order. It makes it easier to verify that something is imported, and avoids duplicated imports.
- Use absolute imports only. This is compatible across a wider range of Python versions, and avoids confusion about what is being imported.
- Avoid renaming imported modules. This makes code more difficult to re-use, and is not grep-friendly.

```
# Good.
from theano import tensor
# Bad.
from theano import tensor as T
```

- Avoid using lists if all you care about is iterating on something. Using lists:
  - uses more memory (and possibly more CPU if the code may break out of the iteration),
  - can lead to ugly code when converted to Python 3 with 2to3,
  - can have a different behavior if evaluating elements in the list has side effects (if you want these side effects, make it explicit by assigning the list to some variable before iterating on it).

Iterative version	List version

`my_dict.iterkeys`
`my_dict.itervalues`
`my_dict.iteritems`
	`my_dict.keys`
	`my_dict.values`
	`my_dict.items`

`itertools.ifilter`
`itertools.imap`
`itertools.izip`
	`filter`
	`map`
	`zip`

`xrange`	`range`

Code example with `map`:

```
# Good.
for f_x in imap(f, x):
    ...
all_f_x = map(f, x)
map(f, x)   # f has some side effect.
# Bad.
for element in map(f, x):
```

```
    ...
imap(f, x)
```

- Generally prefer list comprehensions to `map` / `filter`, as the former are easier to read.

```
# Good.
non_comments = [line.strip() for line in my_file.readlines()
                              if not line.startswith('#')]
# Bad.
non_comments = map(str.strip,
                   ifilter(lambda line: not line.startswith('#'),
                           my_file.readlines()))
```

- Use `in` on container objects instead of using class-specific methods: it is easier to read and may allow you to re-use your code with different container types.

```
# Good.
has_key = key in my_dict
has_substring = substring in my_string
# Bad.
has_key = my_dict.has_key(key)
has_substring = my_string.find(substring) >= 0
```

- Do not use mutable arguments as default values. Instead, use a helper function (conditional expressions are forbidden at this point, see below).

```
# Good.
def f(array=None):
    array = pylearn.if_none(array, [])
    ...
# Bad.
def f(array=[]): # Dangerous if `array` is modified down the road.
    ...
```

- All top-level classes should inherit from `object`. It makes some 'under-the-hood' differences that can be very useful for Python black magic adepts.

```
# Good.
class MyClass(object):
    pass
# Bad.
class MyClass:
    pass
```

- Always raise an exception with `raise MyException(args)` where `MyException` inherits from `Exception`. This is required for compatibility across all versions of Python.

```
# Good.
raise NotImplementedError('The Pylearn team is too lazy.')
# Bad.
raise NotImplementedError, 'The Pylearn team is too lazy.'
raise 'The Pylearn team is too lazy to implement this.'
```

- Use a leading underscore '_' in names of internal attributes / methods, but avoid the double underscore '__' unless you know what you are doing.

Additional Recommendations

Things you should do even if they are not listed in official guidelines:

- All Python code files should start like this:

```
"""Module docstring as the first line, as usual."""

__authors__   = "Olivier Delalleau, Frederic Bastien, David Warde-Farley"
__copyright__ = "(c) 2010, Universite de Montreal"
__license__   = "3-clause BSD License"
__contact__   = "Name Of Current Guardian of this file <email@address>"
```

- Use `//` for integer division and `/ float(...)` if you want the floating point operation (for readability and compatibility across all versions of Python).

```
# Good.
n_samples_per_split = n_samples // n_splits
mean_x = sum(x) / float(len(x))
# Bad.
n_samples_per_split = n_samples / n_splits
mean_x = sum(x) / len(x)
```

- If you really have to catch all exceptions, in general you should use `except Exception:` rather than `except:`, as the latter also catches interrupts like when hitting Ctrl-C.

```
# Good (assuming you *must* be catching all exceptions).
try:
    someting_that_may_fail_in_unexpected_ways()
except Exception:
    do_something_if_it_failed()
# Bad.
try:
    someting_that_may_fail_in_unexpected_ways()
except:
    do_something_if_it_failed()
```

- Use either `try ... except` or `try ... finally`, but do not mix `except` with `finally` (which is not supported in Python 2.4). You can however embed one into the other to mimic the `try ... except ... finally` behavior.

```
# Good.
try:
    try:
        something_that_may_fail_with_some_known_error()
    except SomeError:
        do_something_if_it_failed()
finally:
    always_do_this_regardless_of_what_happened()
# Bad.
try:
    something_that_may_fail_with_some_known_error()
except SomeError:
    do_something_if_it_failed()
```

```
    finally:
        always_do_this_regardless_of_what_happened()
```

- No conditional expression (not supported in Python 2.4). These are expressions of the form `x = y if condition else z`.
- Do not use the `all` and `any` builtin functions (they are not supported in Python 2.4). Instead, import them from `theano.gof.python25` (or use `numpy.all` / `numpy.any` for array data).
- Do not use the `hashlib` module (not supported in Python 2.4). We will probably provide a wrapper around it to be compatible with all Python versions.
- Use `numpy.inf` and `numpy.nan` rather than `float('inf')` / `float('nan')` (should be slightly more efficient even if efficiency is typically not an issue here, the main goal being code consistency). Also, always use `numpy.isinf` / `numpy.isnan` to test infinite / NaN values. This is important because `numpy.nan != float('nan')`.
- Whenever possible, mimic the numpy / scipy interfaces when writing code similar to what can be found in these packages.
- Avoid backslashes whenever possible. They make it more difficult to edit code, and they are ugly (as well as potentially dangerous if there are trailing white spaces).

```
    # Good.
    if (cond_1 and
        cond_2 and
        cond_3):

        # Note that we added a blank line above to avoid confusion between
        # conditions and the rest of the code (this would not have been
        # needed if they were at significantly different indentation levels).
        ...
    # Bad.
    if cond_1 and \
       cond_2 and \
       cond_3:

        ...
```

- When indenting multi-line statements like lists or function arguments, keep elements of the same level aligned with each other. The position of the first element (on the same line or a new line) should be chosen depending on what is easiest to read (sometimes both can be ok). Other formattings may be ok depending on the specific situation, use common sense and pick whichever looks best.

```
    # Good.
    for my_very_long_variable_name in [my_foo, my_bar, my_love,
                                       my_everything]:
        ...
    for my_very_long_variable_name in [
            my_foo, my_bar, my_love, my_everything]:
        ...
    # Good iff the list needs to be frequently updated or is easier to
    # understand when each element is on its own line.
    for my_very_long_variable_name in [
```

```
            my_foo,
            my_bar,
            my_love,
            my_everything,
            ]:
        ...
    # Good as long as it does not require more than two lines.
    for my_very_long_variable_name in [my_foo,
                                       my_bar]:
        ...
    # Bad.
    for my_very_long_variable_name in [my_foo, my_bar, my_love,
            my_everything]:
        ...
    for my_very_long_variable_name in [my_foo,
                                       my_bar,
                                       my_love,
                                       my_everything]:
        ...
```

- Use the `key` argument instead of `cmp` when sorting (for Python 3 compatibility).

```
    # Good.
    my_list.sort(key=abs)
    # Bad.
    my_list.sort(cmp=lambda x, y: cmp(abs(x), abs(y)))
```

- Whenever you read / write binary files, specify it in the mode ('rb' for reading, 'wb' for writing). This is important for cross-platform and Python 3 compatibility (e.g. when pickling / unpickling objects).

```
    # Good.
    cPickle.dump(obj, open('my_obj.pkl', 'wb', protocol=-1))
    # Bad.
    cPickle.dump(obj, open('my_obj.pkl', 'w', protocol=-1))
```

- Avoid tuple parameter unpacking as it can lead to very ugly code when converting to Python 3.

```
    # Good.
    def f(x, y_z):
        y, z = y_z
        ...
    # Bad.
    def f(x, (y, z)):
        ...
```

- Only use `cPickle`, not `pickle` (except for debugging purpose since error messages from `pickle` are sometimes easier to understand).
- A script's only top-level code should be something like:

```
    if __name__ == '__main__':
        sys.exit(main())
```

- Avoid isinstance(x, str) as this don't work with unicode string, use:

```
    isinstance(x, basestring)
```

7.13.3 The `logging` Module vs. the `warning` Module

No print statement should be used. Instead please use logging module. For mode details look at here.

The `logging` Module

A central logging facility for Python capable of logging messages of various categories/urgency and choosing with some granularity which messages are displayed/suppressed, as well as where they are displayed or written. This includes an `INFO` level for innocuous status information, a `WARNING` level for unexpected state that is still recoverable, `DEBUG` for detailed information which is only really of interest when things are going wrong, etc.

In addition to the library documentation, see this helpful tutorial, Python Logging 101.

The `warning` Module

The `warning` module in the standard library and its main interface, the `warn()` function, allows the programmer to issue warnings in situations where they wish to alert the user to some condition, but the situation is not urgent enough to throw an exception. By default, a warning issued at a given line of the code will only be displayed the first time that line is executed. By default, warnings are written to `sys.stderr` but the `warning` module contains flexible facilities for altering the defaults, redirecting, etc.

Which? When?

It is our feeling that the `logging` module's `WARNING` level be used to log warnings more meant for *internal*, *developer* consumption, to log situations where something unexpected happened that may be indicative of a problem but is several layers of abstraction below what a user of the library would care about.

By contrast, the warning module should be used for warnings intended for user consumption, e.g. alerting them that their version of Pylearn is older than this plugin requires, so things may not work as expected, or that a given function/class/method is slated for deprecation in a coming release (early in the library's lifetime, `DeprecationWarning` will likely be the most common case). The warning message issued through this facility should avoid referring to Pylearn internals.

7.13.4 Code Sample

The following code sample illustrates some of the coding guidelines one should follow in Pylearn. This is still a work-in-progress. Feel free to improve it and add more!

```
#! /usr/env/bin python

"""Sample code. Edit it as you like!"""
```

```
__authors__    = "Olivier Delalleau"
__copyright__ = "(c) 2010, Universite de Montreal"
__license__    = "3-clause BSD License"
__contact__    = "Olivier Delalleau <delallea@iro>"

# Standard library imports are on a single line.
import os, sys, time

# Third-party imports come after standard library imports, and there is
# only one import per line. Imports are sorted lexicographically.
import numpy
import scipy
import theano
# Individual 'from' imports come after packages.
from numpy import argmax
from theano import tensor

# Application-specific imports come last.
# The absolute path should always be used.
from pylearn import datasets, learner
from pylearn.formulas import noise


# All exceptions inherit from Exception.
class PylearnError(Exception):
    # TODO Write doc.
    pass

# All top-level classes inherit from object.
class StorageExample(object):
    # TODO Write doc.
    pass


# Two blank lines between definitions of top-level classes and functions.
class AwesomeLearner(learner.Learner):
    # TODO Write doc.

    def __init__(self, print_fields=None):
        # TODO Write doc.
        # print_fields is a list of strings whose counts found in the
        # training set should be printed at the end of training. If None,
        # then nothing is printed.
        # Do not forget to call the parent class constructor.
        super(AwesomeLearner, self).__init__()
        # Use None instead of an empty list as default argument to
        # print_fields to avoid issues with mutable default arguments.
        self.print_fields = if_none(print_fields, [])

    # One blank line between method definitions.
    def add_field(self, field):
        # TODO Write doc.
        # Test if something belongs to a container with `in`, not
```

```
        # container-specific methods like `index`.
        if field in self.print_fields:
            # TODO Print a warning and do nothing.
            pass
        else:
            # This is why using [] as default to print_fields in the
            # constructor would have been a bad idea.
            self.print_fields.append(field)

    def train(self, dataset):
        # TODO Write doc (store the mean of each field in the training
        # set).
        self.mean_fields = {}
        count = {}
        for sample_dict in dataset:
            # Whenever it is enough for what you need, use iterative
            # instead of list versions of dictionary methods.
            for field, value in sample_dict.iteritems():
                # Keep line length to max 80 characters, using parentheses
                # instead of \ to continue long lines.
                self.mean_fields[field] = (self.mean_fields.get(field, 0) +
                                           value)
                count[field] = count.get(field, 0) + 1
        for field in self.mean_fields:
            self.mean_fields[field] /= float(count[field])
        for field in self.print_fields:
            # Test is done with `in`, not `has_key`.
            if field in self.sum_fields:
                # TODO Use log module instead.
                print '%s: %s' % (field, self.sum_fields[field])
            else:
                # TODO Print warning.
                pass

    def test_error(self, dataset):
        # TODO Write doc.
        if not hasattr(self, 'sum_fields'):
            # Exceptions should be raised as follows (in particular, no
            # string exceptions!).
            raise PylearnError('Cannot test a learner that was not '
                               'trained.')
        error = 0
        count = 0
        for sample_dict in dataset:
            for field, value in sample_dict.iteritems():
                try:
                    # Minimize code into a try statement.
                    mean = self.mean_fields[field]
                # Always specicy which kind of exception you are
                # intercepting with except.
                except KeyError:
                    raise PylearnError(
                        "Found in a test sample a field ('%s') that had "
```

```
                        "never been seen in the training set." % field)
                error += (value - self.mean_fields[field])**2
                count += 1
        # Remember to divide by a floating point number unless you
        # explicitly want an integer division (in which case you should
        # use //).
        mse = error / float(count)
        # TODO Use log module instead.
        print 'MSE: %s' % mse
        return mse


def if_none(val_if_not_none, val_if_none):
    # TODO Write doc.
    if val_if_not_none is not None:
        return val_if_not_none
    else:
        return val_if_none


def print_subdirs_in(directory):
    # TODO Write doc.
    # Using list comprehension rather than filter.
    sub_dirs = sorted([d for d in os.listdir(directory)
                       if os.path.isdir(os.path.join(directory, d))])
    print '%s: %s' % (directory, ' '.join(sub_dirs))
    # A `for` loop is often easier to read than a call to `map`.
    for d in sub_dirs:
        print_subdirs_in(os.path.join(directory, d))


def main():
    if len(sys.argv) != 2:
        # Note: conventions on how to display script documentation and
        # parse arguments are still to-be-determined. This is just one
        # way to do it.
        print("""\
Usage: %s <directory>
For the given directory and all sub-directories found inside it, print
the list of the directories they contain."""
              % os.path.basename(sys.argv[0]))
        return 1
    print_subdirs_in(sys.argv[1])
    return 0


# Top-level executable code should be minimal.
if __name__ == '__main__':
    sys.exit(main())
```

7.13.5 Automatic Code Verification

Tools will be available to make it easier to automatically ensure that code committed to Pylearn complies to above specifications. This work is not finalized yet, but David started a Wiki page with helpful configuration tips for Vim.

7.13.6 Commit message

- A one line summary. Try to keep it short, and provide the information that seems most useful to other developers: in particular the goal of a change is more useful than its description (which is always available through the changeset patch log). E.g. say "Improved stability of cost computation" rather than "Replaced log(exp(a) + exp(b)) by a * log(1 + exp(b -a)) in cost computation".
- If needed a blank line followed by a more detailed summary
- **Make a commit for each logical modification**
  - This makes reviews easier to do
  - This makes debugging easier as we can more easily pinpoint errors in commits with hg bisect
- NEVER commit reformatting with functionality changes
- **Review your change before commiting**
  - "hg diff <files>..." to see the diff you have done
  - "hg record" allows you to select which changes to a file should be committed. To enable it, put into the file ~/.hgrc:

```
[extensions]
hgext.record=
```

  - hg record / diff force you to review your code, never commit without running one of these two commands first
- **Write detailed commit messages in the past tense, not present tense.**
  - Good: "Fixed Unicode bug in RSS API."
  - Bad: "Fixes Unicode bug in RSS API."
  - Bad: "Fixing Unicode bug in RSS API."
- Separate bug fixes from feature changes.
- **When fixing a ticket, start the message with "Fixed #abc"**
  - Can make a system to change the ticket?
- **When referencing a ticket, start the message with "Refs #abc"**
  - Can make a system to put a comment to the ticket?

7.13.7 Theano

Theano used many standard for its docstring. I think we should consentrate on the more frequent: http://sphinx.pocoo.org/markup/desc.html#info-field-lists.

7.13.8 UTF-8

To make a file UTF-8 compatible, just add this line at the begining of the file:

```
# -*- coding: utf-8 -*-
```

7.13.9 TODO

Things still missing from this document, being discussed in coding_style.txt:

- Proper style for C code
- Enforcing 100% test coverage of the code base
- Providing ways to add type checking for function arguments
- Conventions for script usage documentation and argument parsing
- Conventions for class / method / function documentation
- Guidelines for serialization-friendly code (hint: nested and lambda functions, as well as instance methods, cannot be serialized, and apparently there are some issues with decorators – to be investigated).

7.14 Overview

This page gives a high-level overview of the Pylearn2 library and describes how the various parts fit together.

First, before learning Pylearn2 it is imperative that you have a good understanding of Theano. Before learning Pylearn2 you should first understand:

- How Theano uses Variables, Ops, and Apply nodes to represent symbolic expressions.
- What a Theano function is.
- What Theano shared variables are and how they can make state persist between calls to Theano functions.

Once you have that under your belt, we can move on to Pylearn2 itself. Note that throughout this page we will mention several different classes and functions but not completely describe their parameters. The purpose of this page is to give you a basic idea of what is possible in Pylearn2, what kind of object is needed to accomplish various kinds of tasks, and where these objects are located. To find out exactly what each object or function does, you should read its docstring in the python code itself. As always, e-mail pylearn-dev@googlegroups.com if you find the documentation lacking, confusing, or wrong in any way.

7.14.1 The Model class

A good central starting point is the Model, defined in pylearn2.models.model.py. Most of Pylearn2 is designed either to train a Model or to do some analysis on a trained Model (evaluate its classification accuracy, etc.)

A Model can be almost anything. It can be a probabilistic model that generates data. It can be a deterministic model that maps from inputs to outputs, like a support vector machine. It can be as simple as a single Gaussian distribution or as complex as a deep Boltzmann machine. The main defining feature of a Models is just that it has a set of parameters represented as theano shared variables.

7.14.2 Training a Model while avoiding Pylearn2 entanglements

Part of the Pylearn2 vision is that users should be able to use only the pieces of the library that they want to. This way you don't need to learn the entire library in order to make use of it. If you want to learn about as little of Pylearn2 as possible, you can simply implement two methods:

- `train_all(dataset)`: This method is passed a Pylearn2 Dataset. When called, it should do one pass through the dataset and update the parameters accordingly. For example, k-means could assign every point in the dataset to a centroid, then update the centroid to the mean of all the points belong to that centroid.
- `continue_learning()`: This method should return True if the algorithm needs to continue learning. For example, the k-means algorithm could return True if the centroids moved more than a certain amount on the most recent pass. (If not trained at all, this should always return True).

We'll say more about what a Pylearn2 Dataset is later. For now, you just need to know that it is a collection of data examples.

If you have a Model and a Dataset you can train them from your own python script like this:

```
while model.continue_learning():
    model.train_all(dataset)
```

This approach to training is simple and easy, and still allows for a lot of powerful functionality. We could imagine a more advanced while loop that inspects or modifies the model after each call to train_all (and indeed, this what the Train object does). However, it violates some of the key goals of the Pylearn2 Vision.

Suppose that the Model is an RBM with Gaussian visible units. There are several ways to train such RBMs. They can be trained with contrastive divergence, persistent contrastive divergence, score matching, denoising score matching, and noise contrastive estimation. Trying to capture all of these with a "train_all" method is a fool's errand. In Pylearn2, we prefer to factor learning algorithms into many different pieces. If you don't want to work with the entire Pylearn2 ecosystem, train_all is an acceptable way to implement learning in your model. However, an ideal Pylearn2 Model will contain very little functionality of its own. We will later describe how to use TrainingAlgorithms and Costs to make a Model that can be trained in many different ways.

First, however, we'll describe how to train a Model using the main "control center" of Pylearn2: the Train object.

7.14.3 Training a Model with the Train object

In the previous section, we described how to train a Model with a simple python while loop. Pylearn2 provides a convenience class for training Models using several powerful additional features. This is the pylearn2.train.Train class.

At its bare minimum, the Train class takes a dataset and a model in its constructor. Calling the class's main_loop method then makes it do the same thing as the simple while loop:

```
# This function...
def train(dataset, model):
    while model.continue_learning():
        model.train_all(dataset)

# does the same thing as this function...
def train(dataset, model):
    loop = Train(dataset, model)
    loop.main_loop()
```

The Train class makes it easy to use several of Pylearn2's other more powerful features. For example, by specifying the save_path and save_freq arguments, the Train object can save your model after each epoch (an "epoch" is a call to train_all). The save can be configured not to overwrite pre-existing files. Also, note that on the second iteration requiring a save and all subsequent such iterations, the Train object will need to overwrite a file that it made earlier. When doing so, it will make a backup of the file that it removes if the write succeeds. If your job dies in the middel of the write, you can still recover the backup.

These are just a few examples of the extra convenient features that come from using the pylearn2 Train object. Later we will describe how to make it more powerful using callbacks and training algorithm plugins.

7.14.4 Configuring the Train object with a YAML file

Pylearn2 has to solve many problems that are not directly related to machine learning and as a result, not all that interesting to Pylearn2's developers, who are a bunch of machine learning grad students.

One of these problems is how to associate a Model with a polymorphic Dataset object and have this association persist after the Model has been serialized, for example to disk. Usually Datasets are big enough that we don't want to serialize a whole copy of the Dataset with every Model we save. But we preprocess Datasets enough that we don't want to store every possible preprocessed dataset and give each model a hard link to the preprocessed dataset.

Instead, we currently solve this problem by using the YAML markup language to store a description of how to make the Dataset in the Model. This is not necessarily an ideal solution, and if you have a better idea and some willingness to implement it, please let us know.

First, a brief explanation of why you want to associate a Dataset with a Model. Suppose your model is a classifer that works on the MNIST dataset. After training the Model, you might want to look at the weights it has learned. MNIST consists of 28 x 28 pixel greyscale images, but the weights are probably stored as 784-element vectors. In order for Pylearn2's visualization functionality to render the weights correctly as an image, it will need to know that the model was trained on the MNIST dataset. This is just one example of why correct analysis of a saved model requires knowledge of the dataset.

Annotating every Model with a correctly-formatted yaml string specifying the dataset would be an annoying and error-prone thing to do from inside a python script. Instead, we try to make all of our training jobs specifiable via a yaml file. The substring of that file used to describe the dataset is then stored in the Model. If you provide a bad description of the dataset, the job doesn't run, so the description in the Model should always be correct. The only way it can go wrong is if conditions change between when the Model is serialized and when it is deserialized (for example, if the yaml string refers to a file that gets moved, or if you deserialize the model file on a machine that has the dataset file in a different place).

Using YAML for everything is thus not something we would ideally like to have designed into the library, but for now it is the easiest way to make your life convenient.

To run an experiment using only a yaml string, the easiest thing to do is save a yaml file with these contents:

```
!obj:pylearn2.train.Train {
        dataset : FILL IN DESCRIPTION OF DATASET HERE
        model : FILL IN DESCRIPTION OF MODEL HERE
        }
```

You can then run the experiment with this command:

```
train.py my_train_job.yaml
```

Here, train.py is a script stored in pylearn2/scripts. Examples throughout the Pylearn2 documentation will assume that this is in your PATH variable.

7.14.5 The Monitor

The Monitor is an object that monitors training of a Model. If you use the Train object it will be set up and maintained for you automatically, so we won't go into the details of how that works here.

The Monitor keeps track of several "channels" on various datasets. A channel might be something like a classifier's accuracy, in which case the monitor could have one channel called "train_acc" tracking accuracy on the training set and another called "valid_acc" tracking accuracy on a held-out validation set.

The main way that you as a user add channels is by implementing the get_monitoring_channels method of various objects like Models and Costs. You'll need to tell the training algorithm which datasets to monitor on.

It's possible for you to view the channels of a saved model using the plot_monitor.py script. The Monitor is also an important part of many training algorithms because the algorithms have the ability to react to values in the monitor. For example, many training algorithms use "early stopping," where an iterative training method quits after accuracy on some validation set starts to drop, in order to prevent overfitting.

7.14.6 TrainingAlgorithm

A TrainingAlgorithm is an object that interacts with a Model to update the Model's parameters given some data. If you supply a TrainingAlgorithm to the Train object, it will call the TrainingAlgorithm to do each epoch, rather than calling the Model's train_all method.

Pylearn2 includes some basic yet powerful training algorithms, SGD (Stochastic Gradient Descent) and BGD (Batch Gradient Descent). Both of these algorithms work by minizing some cost function of the model

parameters and the training data. For example, to fit a logistic regression classifier, you could minimize -log P(Y | X) where Y is training labels and X is training input features. Both of them are configurable to perform more advanced optimization algorithms. For example, BGD can be configured to do nonlinear conjugate gradient descent. SGD supports user-provided callbacks that can provide powerful extra features. Pylearn2 uses this callback system to implement Polyak averaging.

Both of these training algorithms also accept a TerminationCriterion object that determines when to stop learning. A common practice is for the TerminationCriterion to look at channels in the monitor to determine when a learning algorithm has converged. This is how we implement early stopping as mentioned above.

7.14.7 TrainingCallback

The Train object itself also supports callbacks that run after each iteration. These allow further customization of the training procedure. For example, the OneOverEpoch callback can be used to make the SGD algorithm use a 1/t learning rate schedule instead of a fixed learning rate. You can also pass in user-defined callbacks so the sky is the limit.

7.14.8 Cost

A Cost is an object that describes an objective function. Pylearn2 uses these to control both the SGD and BGD algorithms. A Cost describes the objective function both in terms of its value, given some inputs and a model to provide the parameters, and its gradients. If no custom implementation of the gradients is provided, the default implementation simply uses Theano's symbolic differentiation system to provide the gradients.

However, you can accomplish more powerful functionality by returning other gradients. For example, the log likelihood of an RBM is intractable, but its gradient can be approximate with a sampling procedure. You could implement RBM training with the SGD object by returning None for the objective function value but providing sampling-based approximations to the gradient.

7.14.9 Datasets

One of the key principles of Pylearn2 development is that we make features when we need them. One place where this is pretty obvious is the Dataset section of the library. Most people that actually use pylearn2 use it to train complicated models like RBMs on little image patch datasets like MNIST or CIFAR-10. This is one section of the library you can really expect to grow as we move to datasets that don't fit in memory, use more exotic data types, etc.

For now, there are two basic ways data can be viewed: as a design matrix or as a "topological view." The topological view is basically the true underlying form of the data. A batch of image data is a 4-tensor with one axis indexing into different images, one axis indexing into image channels (red, green, blue), and two topological axes indexing the rows and columns of the image. In the design matrix view, the image is flattened into a row vector of size rows * columns * channels.

Different pieces of pylearn2 functionality may ask the dataset for different views of the same data. The Dataset object should be able to provide either view at any time. You might want to train a densely connected RBM on design matrices, or a convolutional MLP on topological views, or you might just want to display an image of a topological view to the user.

The datasets directory includes a Preprocessor class that can modify a Dataset. This should not be considered a way of implementing view conversion, since a Dataset should always be ready to provide either view of its data rapidly. Preprocessors are powerful and can modify datasets in many ways, including changing their number of examples or what kind of data they store. For example, you might want to preprocess the CIFAR-10 dataset of 50,000 32x32 images into a dataset of 200,000 6x6 image patches.

Many common preprocessing operations can be easily represented by Blocks, a kind of Pylearn2 class that takes a batch of input examples, processes them independently, and returns a batch of output examples. To avoid separately implementing the same operation as both a Block and a Preprocessor, there is a generic BlockPreprocessor that can map a Block over a whole Dataset.

Preprocessors are used when you have an expensive operation that you want to do once and save. If you have cheap preprocessing that you would like to do on the fly for each batch, you can do this with the TransformerDataset.

Datasets should be considered relatively abstract entities; in particular we should not assume that they contain a finite number of examples, or that jumping to a particular example index is easy. For example, consider the dataset of small patches drawn from a collection of large, irregularly sized images. Due to the irregular size of the images we need to know the size of each of them in advance if we want to jump to patch number 543,286,932. pylearn2.utils.iteration provides interfaces for abstractly iterating through examples. It supports iteration schemes such as continually drawing random examples from an endless stream.

7.14.10 Spaces and LinearTransforms

Many of our models can be described largely in terms of linear transformations from one space to another. Often changing the exact type of linear transformation can be a useful modeling operation. For example, to scale RBMs to work on large images, we can replace all the matrix multiplies in the model description with convolutions. Pylearn2 uses the TheanoLinear library to provide abstract LinearTransforms. If you write your model in terms of transforming from one Pylearn2 Space to another using a LinearTransform, then your model can easily be converted between dense, convolutional, tiled convolutional, locally connected without weight sharing, etc. just by plugging in the right LinearTransform with compatible Spaces.

7.14.11 Analyzing saved models

A variety of scripts let you analyze your saved models:

- plot_monitor.py lets you plot channels recorded in the model's monitor
- print_monitor.py prints out the final value of each channel in the monitor.
- summarize_model.py prints out a few statistics of the model's parameters
- show_weights.py displays a visualization of the weights of your model's first layer.

7.15 Working with large datasets in Pylearn2

By default Pylearn2 loads all the dataset to the main memory (not GPU). This could be problematic for large datasets. There exists multiple Python/Numpy solutions for dealing with large data:

- Numpy.memmap
- Pytables
- h5py

Pylearn2 currently only supports Pytables and h5py. (memmap support has been introduced in the latest version of Theano. But it has not been tested with Pylearn2 yet.)

7.15.1 PyTables

`pylearn2.datasets.dense_design_matrix.DenseDesignMatrixPyTables` is designed to mimic the behaviour of DenseDesignMatrix but underneath it stores the data in PyTables hdf5 file format. `pylearn2.datasets.svhn.SVHN` is a good example of how to make a DenseDesignMatrixPyTables object and store your data in it.

7.15.2 h5py

If you have your data already saved in hdf5 format, you can use `pylearn2.datasets.hdf5.HDF5Dataset` class to access your data in Pylearn2. For an example of how to save data in hdf5 format and load it with HDF5Dataset, take a look at `pylearn2.datasets.tests.test_hdf5.TestHDF5Dataset`.

7.15.3 PyTables VS h5py

Each library has its own comparison:

- PyTables FAQ
- h5py FAQ

One advantage of h5py over PyTables is that one can use hdf5 files made with other libraries, whereas PyTables hdf5 files are not standard. PyTables also adds some performance-enhancing features and supports LZO and bzip2 compression in addition to zlib (h5py supports gzip and LZF out-of-the-box).

7.15.4 Known issues

- Both hdf5 based solutions are know to crash when the data is accessed in a random order. To avoid this issue, we suggest to use one of the 'sequential' or 'batchwise_shuffled_sequential' iterator schemes.
- Writing large amount of data to hdf5 at once is been know to result in crash. So it's advised to use mini-batches to write the data to files. Some of the prepossessing functions has mini-batch options, but not all of them.
- Users should be aware that any changes to the data will be saved to the data on disk (except in cases where HDF5Dataset is used with *load_all=True*).

7.16 Your models in Pylearn2

7.16.1 Who should read this

We recommend you spend some time with Pylearn2 and read some of our other tutorials before starting with this minimalistic technique. If you are completely new to Pylearn2, have a look at the softmax regression tutorial.

Pylearn2 is great for many things; we'll highlight two here.

- It allows you to experiment with new ideas without much implementation overhead. The library was built to be modular, and it aims to be usable without an extensive knowledge of the codebase. Writing a new model from scratch is usually pretty fast once you know what to do and where to look.
- It has an interface (YAML) that allows one to decouple implementation from experimental choices, enabling experiments to be constructed in a light and readable fashion.

Obviously, there is always a trade-off between being user-friendly and being flexible, and Pylearn2 is no exception. For instance, users looking for a way to work with sequential data might have a harder time getting started (although we're working to make this experience better).

In this post, we will assume that you have built a regression or classification model with Theano and that the training data can be cast into two matrices, one for training examples and one for training targets. People with different requirements may need to work a little more (e.g. by figuring out how to put their data inside Pylearn2). This tutorial contains useful information for anyone interested in porting a model to Pylearn2.

7.16.2 How is Pylearn2 used?

While many researchers use Pylearn2 as their primary research tool, this doesn't necessarily mean they know or use every feature in Pylearn2. In fact, you can prototype new models in a very Theano-like fashion: write a model as a big monolithic block of hard coded Theano expressions, and wrap that up in the minimal amount of code necessary to be able to plug a model into Pylearn2. **This bare minimum is what we'll explain here.**

The resulting model may be hard to extend, but it represents a good starting point. As you explore new ideas and change the code, you can gradually make it more flexible: a hard coded input dimension gets factored out as a constructor argument, functions being composed are separated into layers, etc.

Our point: **it is alright to stick to the bare minimum when developing a model for Pylearn2**. Your code probably won't satisfy any other use cases than your own, but this is something that you can change gradually as you go. There's no need to overcomplicate things when you start.

7.16.3 The bare minimum

Let's look at that *bare minimum*. It involves writing exactly two subclasses:

- One subclass of *pylearn2.costs.cost.Cost*
- One subclass of *pylearn2.models.model.Model*

Need more than that? Nope. That's it! Let's have a look.

It all starts with a cost expression

In the scenario we're describing, your model maps an input to an output, the output is compared with some ground truth using some measure of dissimilarity, and the parameters of the model are changed to reduce this measure using gradient information.

It is therefore natural that the object that interfaces between the model and the training algorithm represents a cost. The base class for this object is *pylearn2.costs.cost.Cost* and does three main things:

- It describes what data it needs to perform its duty and how it should be formatted.
- It computes the cost expression by feeding the input to the model and receiving its output.
- It differentiates the cost expression with respect to the model parameter and returns the gradients to the training algorithm.

What's nice about *Cost* is if you follow the guidelines we're about to describe, you only have to worry about the cost expression; the gradient part is all handled by the *Cost* base class, and a very useful *DefaultDataSpecsMixin* mixin subclass is defined to handle the data description part (more about that when we look at the *Model* subclass).

Let's look at how the subclass should look:

```
from pylearn2.costs.cost import Cost, DefaultDataSpecsMixin


class MyCostSubclass(Cost, DefaultDataSpecsMixin):
    # Here it is assumed that we are doing supervised learning
    supervised = True

    def expr(self, model, data, **kwargs):
        space, source = self.get_data_specs(model)
        space.validate(data)

        inputs, targets = data
        outputs = model.some_method_for_outputs(inputs)
        loss = # some loss measure involving outputs and targets
        return loss
```

The *supervised* class attribute is used by *DefaultDataSpecsMixin* to know how to specify the data requirements. If it is set to *True*, the cost will expect to receive inputs and targets, and if it is set to *False*, the cost will expect to receive inputs only. In the example, it is assumed that we are doing supervised learning, so we set *supervised* to *True*.

The first two lines of *expr* do some basic input checking and should always be included at the beginning of your *expr* method. Without going too much into detail, *space.validate(data)* will make sure that the data you get is the data you requested (e.g. if you do supervised learning, you need an input a tensor variable and a target tensor variable). How to determine "what you need" will be covered when we look at the *Model* subclass.

In that case, *data* is a tuple containing the inputs as the first element and the targets as the second element.

We then get the model output by calling its *some_method_for_outputs* method, whose name and behaviour is really for you to decide, as long as your *Cost* subclass knows which method to call on the model.

Finally, we compute some loss measure on *outputs* and *targets* and return that as the cost expression.

Note that things don't have to be *exactly* like this. For instance, you could ask the model to have a method that takes inputs and targets as arguments and returns the loss directly, and that would be perfectly fine. All you need is some way to make your *Model* and *Cost* subclasses work together to produce a cost expression in the end.

Defining the model

Now it's time to make things more concrete by writing the model itself. The model will be a subclass of *pylearn2.models.model.Model*, which is responsible for the following:

- Defining what its parameters are
- Defining what its data requirements are
- Doing something with the input to produce an output

As is the case with *Cost*, the *Model* base class does many useful things on its own, provided you set the appropriate instance attributes. Let's have a look at a subclass example:

```
from pylearn2.models.model import Model

class MyModelSubclass(Model):
    def __init__(self, *args, **kwargs):
        super(MyModelSubclass, self).__init__()

        # Some parameter initialization using *args and **kwargs
        # ...
        self._params = [
            # List of all the model parameters
        ]

        self.input_space = # Some `pylearn2.space.Space` subclass
        # This one is necessary only for supervised learning
        self.output_space = # Some `pylearn2.space.Space` subclass

    def some_method_for_outputs(self, inputs):
        # Some computation involving the inputs
```

The first thing you should do if you're overriding the constructor is call the the superclass' constructor. Pylearn2 checks for that and will scold you if you don't.

You should then initialize you model parameters **as shared variables**: Pylearn2 will build an updates dictionary for your model variables using gradients returned by your cost. **Protip: the 'pylearn2.utils.sharedX' method initializes a shared variable with the value and an optional name you provide. This allows your code to be GPU-compatible without putting too much thought into it.** For instance, a weights matrix can be initialized this way:

```
import numpy
from pylearn2.utils import sharedX

self.W = sharedX(numpy.random.normal(size=(size1, size2)), 'W')
```

Put all your parameters in a list as the *_params* instance attribute. The *Model* superclass defines a *get_params* method which returns *self._params* for you, and that is method that is called to get the model parameters when *Cost* is computing the gradients.

Your *Model* subclass should also describe the data format it expects as inputs (*self.input_space*) and the data format of the model's output (*self.output_space*), which is required only if you're doing supervised learning. These attributes should be instances of *pylearn2.space.Space* (and generally are instances of *pylearn2.space.VectorSpace*, a subclass of *pylearn2.space.Space* used to represent batches of vectors). Broadly, this mechanism allows for automatic conversion between different data formats (e.g. if your targets are stored as integer indexes in the dataset but are required to be one-hot encoded by the model).

The *some_method_for_outputs* method is really where all the magic happens. Remember, the name of the method doesn't really matter, as long as your *Cost* subclass knows that it's the one it has to call. This method expects a tensor variable as input and returns a symbolic expression involving the input and its parameters. What happens in between is up to you, and this is where you can put all the Theano code you could possibly hope for, just like you would do in pure Theano scripts.

7.16.4 Examples

Let's demonstrate these ideas by writing two models, one which does supervised learning and one which does unsupervised learning.

The data you train these models on is up to you, as long as it is represented in a matrix of features (each row being an example) and a matrix of targets (where each row is a target for an example). Obviously this second matrix is only required for supervised learning. While this is not the only way to store data in Pylearn2, it is probably the most common method, so we will use it in the remainder of this discussion.

For the purposes of this tutorial, we will train models on the venerable MNIST dataset, which you can download at:

```
wget http://deeplearning.net/data/mnist/mnist.pkl.gz
```

To make things easier to manipulate, we will unzip the archive into six different files:

```
python -c "from pylearn2.utils import serial; \
          data = serial.load('mnist.pkl'); \
          serial.save('mnist_train_X.pkl', data[0][0]); \
          serial.save('mnist_train_y.pkl', data[0][1].reshape((-1, 1))); \
          serial.save('mnist_valid_X.pkl', data[1][0]); \
          serial.save('mnist_valid_y.pkl', data[1][1].reshape((-1, 1))); \
          serial.save('mnist_test_X.pkl', data[2][0]); \
          serial.save('mnist_test_y.pkl', data[2][1].reshape((-1, 1)))"
```

Supervised learning using logistic regression

Let's keep things simple by porting to Pylearn2 the *Hello World!* of supervised learning: logistic regression. For a refresher, we suggest that you first read the deeplearning.net tutorial on logistic regression. Here is what we need to do:

- Implement the negative log-likelihood (NLL) loss in our *Cost* subclass

- Initialize the model parameters W and b
- Implement the model's logistic regression output

Let's start with the *Cost* subclass:

```
import theano.tensor as T
from pylearn2.costs.cost import Cost, DefaultDataSpecsMixin


class LogisticRegressionCost(DefaultDataSpecsMixin, Cost):
    supervised = True

    def expr(self, model, data, **kwargs):
        space, source = self.get_data_specs(model)
        space.validate(data)

        inputs, targets = data
        outputs = model.logistic_regression(inputs)
        loss = -(targets * T.log(outputs)).sum(axis=1)
        return loss.mean()
```

We assumed our model has a *logistic_regression* method which accepts a batch of examples and computes the logistic regression output. We will implement that method in just a moment. We also computed the loss as the average negative log-likelihood of the targets given the logistic regression output, as described in the deeplearning.net tutorial. Also, notice how we set *supervised* to *True*.

Now for the *Model* subclass:

```
import numpy
import theano.tensor as T
from pylearn2.models.model import Model
from pylearn2.space import VectorSpace
from pylearn2.utils import sharedX


class LogisticRegression(Model):
    def __init__(self, nvis, nclasses):
        super(LogisticRegression, self).__init__()

        self.nvis = nvis
        self.nclasses = nclasses

        W_value = numpy.random.uniform(size=(self.nvis, self.nclasses))
        self.W = sharedX(W_value, 'W')
        b_value = numpy.zeros(self.nclasses)
        self.b = sharedX(b_value, 'b')
        self._params = [self.W, self.b]

        self.input_space = VectorSpace(dim=self.nvis)
        self.output_space = VectorSpace(dim=self.nclasses)

    def logistic_regression(self, inputs):
        return T.nnet.softmax(T.dot(inputs, self.W) + self.b)
```

The model's constructor receives the dimensionality of the input and the number of classes. It initializes the weights matrix and the bias vector with *sharedX*. It also sets its input space to an instance of *VectorSpace* of the dimensionality of the input (meaning it expects the input to be a batch of examples which are all vectors of size *nvis*) and its output space to an instance of *VectorSpace* of dimension *nclasses* (meaning it produces an output corresponding to a batch of probability vectors, one element for each possible class).

The *logistic_regression* method does pretty much what you would expect: it returns a linear transformation of the input followed by a softmax non-linearity.

How about we give it a try? Save those two code snippets in a single file (e.g. *log_reg.py*) and save the following in *log_reg.yaml*:

```
!obj:pylearn2.train.Train {
    dataset: &train !obj:pylearn2.datasets.dense_design_matrix.DenseDesignMatrix {
        X: !pkl: 'mnist_train_X.pkl',
        y: !pkl: 'mnist_train_y.pkl',
        y_labels: 10,
    },
    model: !obj:log_reg.LogisticRegression {
        nvis: 784,
        nclasses: 10,
    },
    algorithm: !obj:pylearn2.training_algorithms.sgd.SGD {
        batch_size: 200,
        learning_rate: 1e-3,
        monitoring_dataset: {
            'train' : *train,
            'valid' : !obj:pylearn2.datasets.dense_design_matrix.DenseDesignMatrix {
                X: !pkl: 'mnist_valid_X.pkl',
                y: !pkl: 'mnist_valid_y.pkl',
                y_labels: 10,
            },
            'test' : !obj:pylearn2.datasets.dense_design_matrix.DenseDesignMatrix {
                X: !pkl: 'mnist_test_X.pkl',
                y: !pkl: 'mnist_test_y.pkl',
                y_labels: 10,
            },
        },
        cost: !obj:log_reg.LogisticRegressionCost {},
        termination_criterion: !obj:pylearn2.termination_criteria.EpochCounter {
            max_epochs: 15
        },
    },
}
```

Run the following command:

```
python -c "from pylearn2.utils import serial; \
           train_obj = serial.load_train_file('log_reg.yaml'); \
           train_obj.main_loop()"
```

Congratulations, you just implemented your first model in Pylearn2!

(By the way, the targets you used to initialize 'DenseDesignMatrix' instances were column matrices, yet your model expects to receive one-hot encoded vectors. The reason why you can do that is because Pylearn2 does the conversion for you via the 'data_specs' mechanism. That's why specifying the model's 'input_space' and 'output_space' is important.)

Unsupervised learning using an autoencoder

Let's now have a look at an unsupervised learning example: an autoencoder with tied weights. Once again, we recommend that you read the deeplearning.net tutorial. Here's what we'll do:

- Implement the binary cross-entropy reconstruction loss in our *Cost* subclass
- Initialize the model parameters W and b
- Implement the model's reconstruction logic

Let's start again by the *Cost* subclass:

```
import theano.tensor as T
from pylearn2.costs.cost import Cost, DefaultDataSpecsMixin


class AutoencoderCost(DefaultDataSpecsMixin, Cost):
    supervised = False

    def expr(self, model, data, **kwargs):
        space, source = self.get_data_specs(model)
        space.validate(data)

        X = data
        X_hat = model.reconstruct(X)
        loss = -(X * T.log(X_hat) + (1 - X) * T.log(1 - X_hat)).sum(axis=1)
        return loss.mean()
```

We assumed our model has a *reconstruction* method which encodes and decodes its input. We also computed the loss as the average binary cross-entropy between the input and its reconstruction. This time, however, we set *supervised* to *False*.

Now for the *Model* subclass:

```
import numpy
import theano.tensor as T
from pylearn2.models.model import Model
from pylearn2.space import VectorSpace
from pylearn2.utils import sharedX


class Autoencoder(Model):
    def __init__(self, nvis, nhid):
        super(Autoencoder, self).__init__()

        self.nvis = nvis
        self.nhid = nhid
```

```
        W_value = numpy.random.uniform(size=(self.nvis, self.nhid))
        self.W = sharedX(W_value, 'W')
        b_value = numpy.zeros(self.nhid)
        self.b = sharedX(b_value, 'b')
        c_value = numpy.zeros(self.nvis)
        self.c = sharedX(c_value, 'c')
        self._params = [self.W, self.b, self.c]

        self.input_space = VectorSpace(dim=self.nvis)

    def reconstruct(self, X):
        h = T.tanh(T.dot(X, self.W) + self.b)
        return T.nnet.sigmoid(T.dot(h, self.W.T) + self.c)
```

The constructor looks quite similar to the logistic regression example, except that this time we don't need to specify the model's output space.

The *reconstruct* method simply encodes and decodes its input.

Let's try to train it. Save the two code snippets in a single file. For instance *autoencoder.py*. Then save the following in *autoencoder.yaml*:

```
!obj:pylearn2.train.Train {
    dataset: &train !obj:pylearn2.datasets.dense_design_matrix.DenseDesignMatrix {
        X: !pkl: 'mnist_train_X.pkl',
    },
    model: !obj:autoencoder.Autoencoder {
        nvis: 784,
        nhid: 200,
    },
    algorithm: !obj:pylearn2.training_algorithms.sgd.SGD {
        batch_size: 200,
        learning_rate: 1e-3,
        monitoring_dataset: {
            'train' : *train,
            'valid' : !obj:pylearn2.datasets.dense_design_matrix.DenseDesignMatrix {
                X: !pkl: 'mnist_valid_X.pkl',
            },
            'test' : !obj:pylearn2.datasets.dense_design_matrix.DenseDesignMatrix {
                X: !pkl: 'mnist_test_X.pkl',
            },
        },
        cost: !obj:autoencoder.AutoencoderCost {},
        termination_criterion: !obj:pylearn2.termination_criteria.EpochCounter {
            max_epochs: 15
        },
    },
}
```

Run the following command:

```
python -c "from pylearn2.utils import serial; \
           train_obj = serial.load_train_file('autoencoder.yaml'); \
           train_obj.main_loop()"
```

7.16.5 What have we gained?

At this point you might be thinking *"There's still boilerplate code to write; what have we gained?"*

The answer is that we gained access to the plethora of scripts, model parts, costs and training algorithms which are built into Pylearn2. You don't have to reinvent the wheel anymore when you wish to train using SGD and momentum. If you want to switch from SGD to BGD, then Pylearn2 makes this is as simple as changing the training algorithm description in your YAML file.

As we pointed out earlier, this demonstrates only the **bare minimum** needed to implement a model in Pylearn2. Nothing prevents you from digging deeper in the codebase and overriding some methods to gain new functionalities.

Here's an example of how a few more lines of code can do a lot for you in Pylearn2.

Monitoring various quantities during training

Let's monitor the classification error of our logistic regression classifier.

To do so, you will have to override *Model*'s *get_monitoring_data_specs* and *get_monitoring_channels* methods. The former specifies what the model needs for its monitoring, and in which format they should be provided. The latter does the actual monitoring by returning an *OrderedDict* mapping string identifiers to their quantities.

Let's look at how it's done. Add the following to *LogisticRegression*:

```
# Keeps things compatible for Python 2.6
from theano.compat.python2x import OrderedDict
from pylearn2.space import CompositeSpace


class LogisticRegression(Model):
    # (Your previous code)

    def get_monitoring_data_specs(self):
        space = CompositeSpace([self.get_input_space(),
                                self.get_target_space()])
        source = (self.get_input_source(), self.get_target_source())
        return (space, source)

    def get_monitoring_channels(self, data):
        space, source = self.get_monitoring_data_specs()
        space.validate(data)

        X, y = data
        y_hat = self.logistic_regression(X)
        error = T.neq(y.argmax(axis=1), y_hat.argmax(axis=1)).mean()

        return OrderedDict([('error', error)])
```

The content of *get_monitoring_data_specs* may look cryptic at first. Documentation for data specs can be found here. All you really need to know, is that this is the standard method in Pylearn2 to request a tuple whose first element represents features and second element represents targets.

The content of *get_monitoring_channels* should more familiar. We start by checking *data* just as in *Cost* subclasses' implementation of *expr*, and we separate *data* into features and targets. We then get predictions by calling *logistic_regression* and computing the average error the standard way. We return an *OrderedDict* mapping *'error'* to the Theano expression for the classification error.

If we launch training again using

```
python -c "from pylearn2.utils import serial; \
           train_obj = serial.load_train_file('log_reg.yaml'); \
           train_obj.main_loop()"
```

then you'll see the classification error being displayed with the other monitored quantities.

7.16.6 What's next?

The examples given in this tutorial are obviously very simplistic and could be easily replaced by existing parts of Pylearn2. However, they show a path that one can take to implement arbitrary ideas in Pylearn2.

In order to avoid reinventing the wheel, it is often useful to dig into Pylearn2's codebase to see what has already been implemented. For example, the VAE framework relies on the MLP framework to represent the mapping from inputs to conditional distribution parameters.

While it is often desirable to reuse code, the inherent difficulty of this depends on your knowledge of Pylearn2, and also how similar your model is to what is already implemented. You should never feel ashamed to dump Theano code inside a *Model* subclass' method like we showed here. The modularity of your code can be improved gradually, and at your own pace. In the meantime you can still benefit from Pylearn2's features, like human-readable descriptions of experiments, automatic monitoring of various quantities, easily-interchangeable training algorithms, and so on.

7.17 Quick-start example

The directory `pylearn2/scripts/tutorials/grbm_smd/` contains an example of how to train a model using pylearn2.

There are three steps. First, we create a preprocessed dataset and save it to the filesystem. Then we train a model on that dataset using the `train.py` script. Finally we can inspect the model to see how the learning experiment went.

7.17.1 Step 1: Create the dataset

From the grbm_smd directory, run

```
python make_dataset.py
```

This will take a while.

You should read through `make_dataset.py` to understand what it is doing. The file is heavily commented and is basically a small tutorial on pylearn2 datasets.

As a brief summary, this script will load the CIFAR10 database of 32x32 color images. It will extract 8x8 pixel patches from them, run a regularized version of contrast normalization and whitening on them, then save them to `cifar10_preprocessed_train.pkl`.

7.17.2 Step 2: Train the model

You should have `pylearn2/scripts` in your PATH enviroment variable. `pylearn2/scripts/train.py` should have executable permissions.

From the `grbm_smd` directory, run

```
train.py cifar_grbm_smd.yaml
```

This will also take a while.

You should read the yaml file for more information. It is heavily commented and is basically a tutorial on yaml files and training models.

As a high level summary, it will create a file called `cifar_grbm_smd.pkl`. This file will contain a gaussian binary RBM trained using denoising score matching.

7.17.3 Step 3: Inspect the model

`pylearn2/scripts/show_weights.py` and `pylearn2/scripts/plot_monitor.py` should have executable permissions.

From the `grbm_smd` directory, run

```
show_weights.py cifar_grbm_smd.pkl
```

A window containing the filters learned by the RBM will appear. They should mostly be colorless gabor filters, though some will be color patches.

Note that the filters are still a bit grainy. This is because the example script doesn't train for very long. You can modify `cifar_grbm_smd.yaml` to train for longer if you would like to see prettier filters.

Now close that window and run

```
plot_monitor.py cifar_grbm_smd.pkl
```

This will bring up a simple interface that allows us to visualize different quantities that were tracked over the course of training. Try typing `b,L,M` and pushing enter to display a plot of the objective function and reconstruction error over time (with time on the x-axis indexed by batch number). When you're finished, close the figure and enter `q` to quit.

7.18 Using jobman with Pylearn2

Jobman enables you to store experiments, their original hyper-parameters, and some results, in a database. It is useful, for instance, to schedule experiments to be launched later, launch them (for instance, use a cluster to launch them all in parallel), and finally collect some measurements (reconstruction error, classification

error, training time, for instance) and add them in the database. Then, we can use queries on that database to do some analyses of the results.

For the moment, what can be inside the database is limited to simple data types, and things that are not too big in memory: integers, floating-point numbers, strings, and dictionaries of the above (if the keys are strings).

7.18.1 Bases of Jobman

- state dictionary, linked with the DB
- experiment function, exp(state, channel)

See Jobman documentation for installation instructions, and information on how to use it.

7.18.2 pylearn2/jobman interface

Such an `experiment` function is in Pylearn2 `pylearn2/scripts/jobman/experiment.py`, and will use a specially-crafted `state` dictionary, that specifies:

- which experiment (model, trainer, and so on) to instanciate;
- which hyper-parameters exist and which values they should take;
- and finally which results (or outputs) you want to extract from the trained experiment and store in the database.

For the moment, only one such experiment function exists: `pylearn2.scripts.jobman.train_experiment`, that trains a model the same way `scripts/train.py` would. Other functions may be implemented in the future, with the same structure in `state`.

The `state` dictionary

Here's the structure for the `state` dictionary-like object, that represents a single experiment. For your convenience, you can access to `state`'s properties either as items of a dictionary (`state["some_attr"]`) or as attributes of an object (`state.some_attr`).

`state.`**`yaml_template`**
: String value

  This is a string containing a template for the YAML file that will represent your experiment. This template will typically be shared among different experiments, when these experiments differ only by the value of some hyper-parameters.

  Instead of specifying the values of hyper-parameters in the YAML file, you can specify them using the Python substitution syntax. For instance, if one of your hyper-parameters is a learning rate, you can specify the following in your template:

```
    "learning_rate": %(learning_rate)d
```

If you want to specify a whole object as a hyper-parameter (for instance, if you want to try different classes), you can take advantage from the fact that Python's serialization for dictionaries is close to YAML. For instance, you can specify:

```
"termination_criterion": %(term_crit_builder)s %(term_crit_args)s
```

The identifier inside the parentheses is the *name* you give to your hyper-parameter, the letter afterwards identifies its type (here, "s" for string, "d" for decimal number, every type can usually be converted automatically into a string).

These "place-holders" in the template will be replaced by actual values, that are provided in *state.hyper_parameters*.

state.**hyper_parameters**
Dictionary, all keys must be strings.

This dictionary contains the mapping from hyper-parameter names, as used in the yaml_template string, to the actual value you want to give them in that particular experiment.

For instance, for a YAML template like the one above, a possible value of `state.hyper_parameters` could be:

```
{
    "learning_rate": 1e-4,
    "term_crit_builder": "!obj:pylearn2.training_algorithms.sgd.EpochCounter",
    "term_crit_args": {
        "max_epochs": 2
        }
}
```

The resulting YAML file that will be used in your experiment will look like:

```
{
    [...]
    "learning_rate": 1e-4,
    [...]
    "termination_criterion": !obj:pylearn2.training_algorithms.sgd.EpochCounter {
        max_epochs: 2,
    },
}
```

You can achieve the same effect by specifying only one place-holder for a complete YAML object. If your template is:

```
{ "termination_criterion": %(term_crit)s }
```

and you have, in state.term_crit:

```
{
    '__builder__': 'pylearn2.training_algorithms.sgd.EpochCounter',
    'max_epochs': 2
}
```

The special key `'__builder__'` will be used as the constructor function of the object to build, which will result in the same final YAML string:

```
{ "termination_criterion": !obj:pylearn2.training_algorithms.sgd.EpochCounter {
        max_epochs: 2,
    } }
```

When that YAML string gets generated, it will be parsed to build a `train_object`, that will be trained (`train_object.main_loop()` is called), then returned. That object will depend on what is described in the YAML file. Usually, you will want to extract some information from that object, and save that in the database, to be used as some metric. This is done by *state.extract_results*.

`state.`**`extract_results`**

String, containing the name of a function to call on "train_object".

The function named in *state.extract_results* takes one argument, and returns a dictionary. Like *state.hyper_parameters*, this dictionary's keys are strings, and values can be ints, floats, strings or dictionaries (with integer keys, and they can be nested). The returned dictionary will be used as *state.results*. This function has to be in some place where it can be importable.

For instance, say we want to record the best epoch according to some validation error, and the training and validation errors (classification and NLL) at that training epoch. Those will kept in train_obj.model.monitor.channels, so our function may look like:

```
def result_extractor(train_obj):
    import numpy
    channels = train_obj.model.monitor.channels
    best_index = numpy.argmin(channels["valid_nll"].val_record)

    return dict(
        best_epoch=best_index,
        train_nll=channels["train_nll"].val_record[best_index],
        train_classerr=channels["classerr"].val_record[best_index],
        valid_nll=channels["valid_nll"].val_record[best_index],
        valid_classerr=channels["valid_classerr"].val_record[best_index])
```

Note: The channels and their name will depend on the content of your YAML template string.

If that function is placed in pylearn2/scripts/examples/extractor.py, we will specify:

```
state.extract_results = "pylearn2.scripts.examples.extractor.result_extractor"
```

For a working example, you can have a look at `pylearn2/scripts/jobman/tester.py`

`state.`**`results`**

Dictionary, all keys are strings.

This dictionary will be empty initially, but will be filled after the experiment finishes, by the call to *state.extract_results*.

For instance, the result specification above may give the following result:

```
{
    'best_epoch': 47,
    'train_nll': 0.02,
    'train_classerr': 0.0,
```

```
        'valid_nll': 0.05,
        'valid_classerr': 0.03,
    }
```

Having those informations in the database allows you to efficiently search for the model that has the lowest classification error, for instance.

You can see a complete example of a `state` dictionary in `pylearn2/scripts/jobman/tester.py`.

pylearn2/scripts/jobman/experiment.py

train_experiment

Here is an outline of what is really happening inside the train_experiment function.

- The `state.hyper_parameters` dictionary is converted to a YAML-friendly sub-class of dict.
- Place-holders in `state.yaml_template` are replaced by the values contained in `state.hyper_parameters`, to create a YAML string.
- That string is passed to `pylearn2.config.yaml_parse.load`, and a `train_obj` object is returned.
- `train_obj.main_loop()` is called.
- The function whose name is specified in `state.extract_results` is imported, then called on `train_obj`, returning a results dictionary.
- That dictionary is affected to `state.results`
- The function exits, `state` gets synchronized with the database, and the working directory gets rsync'ed.

7.19 Pylearn2 Vision

A brief statement of the Pylearn2 vision is given on the *Welcome* page. Here we explore some points of the vision statement in more detail:

- We build the parts of the library when we need them or shortly before.
  - There is NOT a big programming effort for stuff that we are not sure we will use.
  - Don't over-engineer. This killed Pylearn1. Start with simple interfaces and introduce generality as it is needed. Don't try to make everything fully general from the start.
  - That being said, keep the overall Vision in mind while coding. Try to design interfaces that will be easy to modify in the future to support predictable desired features.
- Pylearn2 is a machine learning toolbox for scientific experimentation.
  - This means it is OK to expect a high level of machine learning sophistication from our users.
  - It also means the framework should not restrict very much what is possible.

  - These are very different design goals / a very different target user from say scikit-learn where everything should have a "fit" method that just works out of the box.

- One goal we used to have but no one seems to actually work is supporting the scikit-learn interface. It could make sense to add support for the the scikit-learn interface to one of our models such as S3C, RBMs, or autoencoders and add it to
  - https://github.com/scikit-learn/scikit-learn/wiki/Related-Projects
- Dataset interface
  - Pylearn2 datasets need to be able to understand topology. Many kinds of data have different kinds of topology: 1-D topology like a measurement collected over time, 2-D topology like images, 3-D topology like videos, etc.
  - Pylearn2 models may use topology in different ways: convolution, tiled convolution, local receptive fields with no tied weights, Toronto's fovea-approximating techniques, etc.
- Support many views for each object. Make it easy for different component play different roles.
  - For example, generative models and datasets both define probability distributions that we should be able to sample from.
- Contain a concise human-readable experiment description language that makes it easy for other people to replicate our exact experiments with others implementations. This should include hyperparameter and other related configuration. (currently we use yaml for this).
- Include algorithms and utilities that are factored as being separate from the model as much as possible. This includes training algorithms, visualization algorithms, model selection algorithms, model composition or averaging techniques, etc.

7.20 F.A.Q.

We are so thrilled that number of Pylearn2 users is growing each day. With growth there is a heavier traffic on the mailing list. Pylearn2 and its mailing list are mainly maintained by graduate students at LISA lab. Each one of us has their own research projects and hopefully some life outside the lab. Our resources are limited and we prefer these limited resources to be invested in fixing real bugs and answering relevant questions. Also please notice that replying to the mailing list is out of courtesy of the pylearn2 developers. Please read below FAQ section before posting a question to the mailing list.

7.20.1 F.A.Q.

1. ***How do I set PYTHONPATH, I'm Python newbie, etc?*** If you do not have much experience with python, Theano, Linux shell, setting environmental variables and other basics, please spend some time learning them. A simple google search usually will answer your question and places like stackoverflow are an excellent resources for these sort of questions.
2. ***I'm getting errors like: Unrecognized environment variable PYLEARN2...*** Take a look at here *here*. for list of pylearn2 environmental variables.

3. ***How do you solve my X machine learning problem?*** As stated here *here*., the library is mainly intended for ML researchers, nonetheless we are happy that the user scope is larger now. But if you have machine learning problem or a research question, places like metaoptimze, Google Plus Machine learning and Deep Learning groups would be the appropriate places to ask. And if you can't find your answers there, perhaps you need to hire a ML consultant.
4. ***My model doesn't work, doesn't seems to be learning, etc.*** Most of the models in ML have a set of hyper-parameters that need to be tuned. The expertise of how to tune those comes from better understanding of the theory and experience. There is some research in this area, search for hyper-parameter optimization in literature. Using the right model for the right data, proper data pre-processing are other factors that should be paid attention to.
5. ***Is model X implemented? Is feature Y supported?*** Search through the documentation pages and source codes. grep and find are your saviors. If nothing comes up, probably it is not developed yet. We will receive your pull request with open arms if you implement what you were looking for.
6. ***Method X has no documentation?*** Pylearn2 is under rapid development to fit the needs of LISA members' research. Sometimes we have shipped a feature fast without documentation. But now there are unit-tests at work that will prevent any new PR without documentation to be merged. Hopefully soon the whole library will be well documented. We always welcome pull requests adding documentation and tutorials and are more likely to answer questions about how to use undocumented methods if you're asking because you're working on a documentation pull request
7. ***I went through all questions here, and still can't find my answer?*** Before posting your question, search the mailing list archive, someone might have answered your question already there.
8. ***Should I write to pylearn-dev or pylearn-users?*** If you have found a bug or want to develop a new feature pyelarn2-dev list would be the right list. All other questions should go to pylearn-users.
9. ***I found a bug, how do I report it?*** We can't read your mind yet, nor do we have access to NSA data. So please follow these instructions before reporting a bug so that we can understand it better and reproduce the error.
    (a) **Are you using the latest version of Pylearn2 and Theano? By latest version we mean,** the latest development version from the git repo. Both Pylearn2 and Theano are under active development, please take a moment to update them and see if the error still exists. *here*.
    (b) **Have you tried to figure out what's the source of the error?** Pylearn2 is not a end-user customer product, it's an open source library, try to dig into soure code and find the cause of the error
    (c) **If you followed steps 1 and 2 and found a real bug, please write down a** detailed procedure for reproducing the error. The code, the command to execute it and the whole error stack trace.
    (d) Please only use plain-text in the email and avoid using formatting or attaching screenshots.
10. ***I have new feature to add, I have fixed a bug, how do I send a pull request?*** Look at *here*.
11. ***I posted a question and haven't heard back for so long!***

(a) Are you sure your question was not covered here?

(b) **Is it near ICML, NIPS or ICLR deadline, or is it summer? LISA members** will be very busy during conference deadlines, and many students do summer internships. So unfortunately during those times it might take longer to answer your questions.

12. ***I have this problem in Windows.*** None of us work in Windows or even have access to it. But feel free to post your question, maybe some of the other users might be able to help you.

13. ***My prof has given us this assignment, could you help me PLEASE, PLEASE, PLEASE*** ^_^ You would be flagged as moderated.

7.21 YAML for Pylearn2

Note: Code for this section is available in `Pylearn2/doc/yaml_tutorial/autoencoder.py`. Since the `Pylearn2/doc` will not typically be on your `PYTHONPATH`, you can run the code using the following command.

```
[user@host]$ PYTHONPATH=/path/to/Pylearn2/doc python autoencoder.py.
```

Pylearn2 makes extensive use of YAML , a human-readable dataset serialization scripting language, for defining an experimental configuration. This allows the end-user to completely specify an experiment (includes all parts of model, dataset and training algorithm specifications), without having to write a single line of python code. Since scores of YAML tutorials can be found online, this tutorial will focus on the specific set of YAML features employed by Pylearn2, in particular custom tags (registered by Pylearn2) which augment the basic YAML syntax to allow for dynamic object creation (with automatic imports), file unpickling, etc.

7.21.1 Introduction

We will start by defining a basic Python object, which we will then use to highlight Pylearn2's YAML functionality. The object in question is a skeleton class for a basic auto-encoder.

```
class AutoEncoder:

    def __init__(self, nvis, nhid, iscale=0.1,
            activation_fn=numpy.tanh,
            params=None):

        self.nvis = nvis
        self.nhid = nhid
        self.activation_fn = activation_fn

        if params is None:
            self.W = iscale * numpy.random.randn(nvis, nhid)
            self.bias_vis = numpy.zeros(nvis)
            self.bias_hid = numpy.zeros(nhid)
        else:
            self.W = params[0]
```

```
            self.bias_vis = params[1]
            self.bias_hid = params[2]

        print self

    def __str__(self):
        rval  = '%s\n' % self.__class__.__name__
        rval += '\tnvis = %i\n' % self.nvis
        rval += '\tnhid = %i\n' % self.nhid
        rval += '\tactivation_fn = %s\n' % str(self.activation_fn)
        rval += '\tmean std(weights) = %.2f\n' % self.W.std(axis=0).mean()
        return rval

    def save(self, fname):
        fp = open(fname, 'w')
        pickle.dump([self.W, self.bias_vis, self.bias_hid], fp)
        fp.close()
```

The above code does nothing more than allocate the model parameters, pretty-printing the model description, as well as pickling the model parameters in its `save` method.

7.21.2 Object Instantiation with !obj

Objects can be fully specified using Pylearn2/YAML using the syntax `!obj:<package>[.<subpackage>]*.<module>.<object>` syntax. For example, the example below can be used to create an AutoEncoder object, with 784 visible and 100 hidden units, with weights initialized from a centered normal distribution and 0.2 standard deviation.

```
!obj:yaml_tutorial.autoencoder.AutoEncoder {
   "nvis": 784,
   "nhid": 100,
   "iscale": 0.2,
}
```

The `!obj` tag does two things. Starting from the top-level package or module, it recursively imports all other sub-packages, until it imports the final module. For this to succeed, the top-level package should be located in one of the directories listed in the `PYTHONPATH` environment variable, i.e. `import <package>` should succeed from a vanilla python shell. Once the import phase is finished, it then proceeds to instantiate an object of type <object>, whose parameters are given in the keyword style syntax.

Assuming this model description was stored in `example1.yaml`, a Python instantiation of this model can be obtained using the `load` method of the `pylearn2.config.yaml_parse` module (shown below).

```
fp = open('example1.yaml')
model = yaml_parse.load(fp)
print model
fp.close()
```

Running the above code yields the following output.

```
AutoEncoder
    nvis = 784
    nhid = 100
    activation_fn = <ufunc 'tanh'>
    mean std(weights) = 0.20
```

7.21.3 Anchors and References

While anchors and references are not particular to Pylearn2's augmented YAML syntax, they are used widely within the library and thus deserve a brief mention in the tutorial. Anchors associate a unique identifier to a given attribute or object defined in a yaml file. This identifier can then be referenced in another section, either to avoid duplicating code or when references to the same object are required. Note that references are limited to anchors within the same yaml file.

For example, the following yaml script could be used to instantiate an auto-encoder with a square weight-matrix.

```
!obj:yaml_tutorial.autoencoder.AutoEncoder {
   "nvis": &nvis 100,
   "nhid": *nvis,
}
```

In the above example, the number of visible units is augmented with an anchor `&nvis`. We can then specify the number of hidden units indirectly through the reference `*nvis`. This can be very useful if we intend on changing the number of features `nvis` frequently (preventing us from having to modify this number in two places). More interestingly, when anchors are applied to objects, as in `&model !obj:yaml_tutorial.autoencoder.AutoEncoder`, subsequent use of `*model` will create a pointer to the AutoEncoder object already instantiated by the anchor.

7.21.4 Dynamic Includes with !import

The `!import` tag is similar to `!obj`, in that it also recursively imports packages, subpackages and modules. It does not however instantiate any object. The end result is thus a pointer to the symbol (package, module or function) specified by the tag. For example, the following allows us to change the non-linearity used by our auto-encoder's hidden layer from the default tanh to a sigmoidal non-linearity.

```
!obj:yaml_tutorial.autoencoder.AutoEncoder {
   "nvis": 784,
   "nhid": 100,
   "iscale": 1.0,
   "activation_fn": !import 'pylearn2.expr.nnet.sigmoid_numpy',
}
```

Assuming the above yaml script is stored in `experiment3.yaml`, running `yaml_load('experiment3.yaml')` yields the following output.

```
AutoEncoder
    nvis = 784
    nhid = 100
```

```
    activation_fn = <function sigmoid_numpy at 0x307a320>
    mean std(weights) = 1.00
```

7.21.5 Loading Files through Pickle

Another useful tag registered by Pylearn2 is `!pkl` which allows the end-user to dynamically load the contents of a pickle file. This can be especially useful when initializing the parameters of a model from a pickled model instance or or to load a dataset stored to disk in pickle format.

To showcase this functionality, we start by re-instantiating the model from the previous configuration (stored in `experiment3.yaml`) and then save the parameters to disk in the file `example3_weights.pkl`.

```
fp = open('experiment3.yaml')
model = yaml_parse.load(fp)
model.save('example3_weights.pkl')
fp.close()
```

Once saved to disk, we can leverage the pickled parameters in a subsequent model as follows:

```
!obj:yaml_tutorial.autoencoder.AutoEncoder {
   "nvis": 784,
   "nhid": 100,
   "iscale": 0.1,
   "params": !pkl: 'example3_weights.pkl',
```

The above is equivalent to constructing an AutoEncoder object as follows:

```
from yaml_tutorial import autoencoder

fp = open('example3_weights.pkl')
params = pickle.load(fp)
fp.close()
model = autoencoder.AutoEncoder(nvis=784, nhid=100, iscale=0.1, params=params)
```

The original yaml (once loaded through `yaml_load`) generates the following output. Note that the mean standard-deviation of the weights is 1.0, despite the iscale parameter being set to 0.1. This is expected because the parameters were initialized from 'example3_weights.pkl', i.e. the pickled parameters of a model initialized with `iscale=1.0`.

```
AutoEncoder
    nvis = 784
    nhid = 100
    activation_fn = <ufunc 'tanh'>
    mean std(weights) = 1.00
```

7.21.6 Putting it all Together

In the same way that `!import` or `!pkl` can be used to define the "value" of a keyword attribute, we can also use the `!obj` tag to instantiate objects as member attributes of another object. Since Pylearn2 experiments are themselves objects of the type `pylearn2.train.Train`, which combine:

- a dataset, of type `pylearn2.datasets.dataset.Dataset`
- a model, of type `pylearn2.models.model.Model`
- a training algorithm, of type `pylearn2.training_algorithms.training_algorithm.TrainingAlgo`

we can specify a Pylearn2 experiment as follows.

```
!obj:pylearn2.train.Train {
    "dataset": !obj:pylearn2.datasets.dense_design_matrix.DenseDesignMatrix &dataset {
        "X" : !obj:numpy.random.normal { 'size':[5,3] },
    },
    "model": !obj:pylearn2.models.autoencoder.DenoisingAutoencoder {
        "nvis" : 3,
        "nhid" : 4,
        "irange" : 0.05,
        "corruptor": !obj:pylearn2.corruption.BinomialCorruptor {
            "corruption_level": 0.5,
        },
        "act_enc": "tanh",
        "act_dec": null,    # Linear activation on the decoder side.
    },
    "algorithm": !obj:pylearn2.training_algorithms.sgd.SGD {
        "learning_rate" : 1e-3,
        "batch_size" : 5,
        "monitoring_dataset" : *dataset,
        "cost" : !obj:pylearn2.costs.autoencoder.MeanSquaredReconstructionError {},
        "termination_criterion" : !obj:pylearn2.termination_criteria.EpochCounter {
            "max_epochs": 1,
        },
    },
    "save_path": "./garbage.pkl"
}
```

This particular example (which can be found in `Pylearn2/pylearn2/scripts/autoencoder_example/dae.ya`) trains a denoising auto-encoder on the `NpyDataset`, using the exhaustive SGD training algorithm. See the pylearn2 documentation for details regarding each argument.

Pylearn2 provides the script `Pylearn2/pylearn2/train.py`. which combines the steps of (1) loading the Train object through `yaml_load` and (2) running multiple iterations of the training algorithm.

```
[user@host]$ cd /path/to/Pylearn2/pylearn2
[user@host]$ train.py scripts/autoencoder_example/dae.yaml
```

7.22 IPython Notebook Tutorials

For more thorough tutorials on different models you can look at the ipython notebook tutorials at "pylearn2/scripts/tutorials/"

7.22.1 How to Run

Here is a tutorial on ipython notebooks. But basically you can move to the tutorial folder and start the ipython notebook engine as follow:

```
$ ipython notebook
```

7.22.2 View Static Versions Online

Here are the links to the static versions of the notebooks that you can view online:

- Softmax Regression
- Multilayer Perceptron
- Convolutional Network
- Stacked Autoencoders

7.23 LICENSE