theano Documentation

Release 0.7

LISA lab, University of Montreal

August 19, 2015

CONTENTS

Theano is a Python library that allows you to define, optimize, and evaluate mathematical expressions involving multi-dimensional arrays efficiently. Theano features:

- **tight integration with NumPy** – Use *numpy.ndarray* in Theano-compiled functions.
- **transparent use of a GPU** – Perform data-intensive calculations up to 140x faster than with CPU.(float32 only)
- **efficient symbolic differentiation** – Theano does your derivatives for function with one or many inputs.
- **speed and stability optimizations** – Get the right answer for `log(1+x)` even when `x` is really tiny.
- **dynamic C code generation** – Evaluate expressions faster.
- **extensive unit-testing and self-verification** – Detect and diagnose many types of mistake.

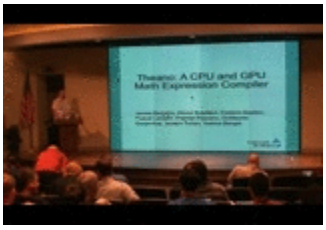
Theano has been powering large-scale computationally intensive scientific investigations since 2007. But it is also approachable enough to be used in the classroom (IFT6266 at the University of Montreal).

CHAPTER

ONE

NEWS

- We added support for `CNMeM` to speed up the GPU memory allocation.
- Theano 0.7 was released 26th March 2015. Everybody is encouraged to update.
- We support cuDNN if it is installed by the user.
- Open Machine Learning Workshop 2014 presentation.
- Colin Raffel tutorial on Theano.
- Ian Goodfellow did a 12h class with exercises on Theano.
- New technical report on Theano: Theano: new features and speed improvements.
- HPCS 2011 Tutorial. We included a few fixes discovered while doing the Tutorial.



You can watch a quick (20 minute) introduction to Theano given as a talk at SciPy 2010 via streaming (or downloaded) video:

> Transparent GPU Computing With Theano. James Bergstra, SciPy 2010, June 30, 2010.

CHAPTER

TWO

DOWNLOAD

Theano is now available on PyPI, and can be installed via `easy_install Theano`, `pip install Theano` or by downloading and unpacking the tarball and typing `python setup.py install`.

Those interested in bleeding-edge features should obtain the latest development version, available via:

```
git clone git://github.com/Theano/Theano.git
```

You can then place the checkout directory on your `$PYTHONPATH` or use `python setup.py develop` to install a `.pth` into your `site-packages` directory, so that when you pull updates via Git, they will be automatically reflected the "installed" version. For more information about installation and configuration, see *installing Theano*.

CHAPTER

THREE

CITING THEANO

If you use Theano for academic research, you are highly encouraged (though not required) to cite the following two papers:

- F. Bastien, P. Lamblin, R. Pascanu, J. Bergstra, I. Goodfellow, A. Bergeron, N. Bouchard, D. Warde-Farley and Y. Bengio. "Theano: new features and speed improvements". NIPS 2012 deep learning workshop. (BibTex)
- J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley and Y. Bengio. "Theano: A CPU and GPU Math Expression Compiler". *Proceedings of the Python for Scientific Computing Conference (SciPy) 2010. June 30 - July 3, Austin, TX* (BibTeX)

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

See our *citation* for details.

CHAPTER

FOUR

DOCUMENTATION

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

- *Installing Theano* – How to install Theano.
- *Theano at a Glance* – What is Theano?
- *Tutorial* – Learn the basics.
- *Library Documentation* – Theano's functionality, module by module.
- *faq* – A set of commonly asked questions.
- *Optimizations* – Guide to Theano's graph optimizations.
- *Extending Theano* – Learn to add a Type, Op, or graph optimization.
- *Developer Start Guide* – How to contribute code to Theano.
- *developer* – Primarily of interest to developers of Theano
- *Internal Documentation* – How to maintain Theano, LISA-specific tips, and more...
- *Release* – How our release should work.
- *Acknowledgements* – What we took from other projects.
- Related Projects – link to other projects that implement new functionalities on top of Theano

You can download the latest PDF documentation, rather than reading it online.

Check out how Theano can be used for Machine Learning: Deep Learning Tutorials.

Theano was featured at SciPy 2010.

CHAPTER

FIVE

COMMUNITY

> "Thank YOU for correcting it so quickly. I wish all packages I worked with would have such an active maintenance - this is as good as it gets :-)"
>
> (theano-users, Aug 2, 2010)

- Register to theano-announce if you want to be kept informed on important change on theano(low volume).
- Register and post to theano-users if you want to talk to all Theano users.
- Register and post to theano-dev if you want to talk to the developers.
- Register to theano-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.
- Ask/view questions/answers at StackOverflow
- We use Github tickets to keep track of issues (however, some old tickets can still be found on Assembla).
- Come visit us in Montreal! Most developers are students in the LISA group at the University of Montreal.

CHAPTER

SIX

HELP!

6.1 How to Seek Help

The appropriate venue for seeking help depends on the kind of question you have.

- How do I? – theano-users mailing list or StackOverflow
- I got this error, why? – theano-users mailing list or StackOverflow (please include the *full* error message, even if it's long)
- I got this error and I'm sure it's a bug – Github ticket
- I have an idea/request – post the suggestion to theano-dev or, even better, implement the idea and submit a GitHub pull request!
- Why do you? – theano-users mailing list (not appropriate for StackOverflow)
- When will you? – theano-dev mailing list (not appropriate for StackOverflow)

Please do take some time to search for similar questions that were asked and answered in the past. If you find something similar that doesn't fully answer your question, it can be helpful to say something like "I found X but it doesn't address facet Y" and link to the previous discussion.

When asking questions on StackOverflow, please use the *theano* tag, so your question can be found, and follow StackOverflow's guidance on asking questions. Consider also using the *python* and *numpy* tags, especially if you are unsure which library your problem relates to.

It's often helpful to include the following details with your question:

- If you have an error, the *full* error message, even if it's long
- Which versions of Python and Theano you're using
- Whether you're using a CPU or GPU device
- Details of your Theano configuration settings (you can print this in Python via print theano.config)

Spending the time to create a minimal specific example of a problem is likely to get you to an answer quicker than posting something quickly that has too much irrelevant detail or is too vague. A minimal example may take you a bit more time to create but the first response is more likely to be the answer you need than, rather than a frustrated request for clarification.

6.2 How to provide help

If you see a question on the theano-users mailing list, or on StackOverflow, that you feel reasonably confident you know an answer to, please do support the community by helping others.

We were all newbies to Theano once and, as the community expands, there is a constant stream of new Theano users looking for help. Perhaps you asked a question when you were first starting out? Now you can pay it forward by helping others. It's also a good way to reinforce your own Theano knowledge.

Often it's easiest to answer a question directly but sometimes it may be better to refer people to a good answer that was provided in the past. Pointing people to relevant sections in the documentation, or to a Theano tutorial, can also be helpful.

When answering questions please be nice (as always!) and, on StackOverflow, follow their guidance for answering questions.

6.2.1 Release Notes

Theano 0.7 (26th of March, 2015)

We recommand to everyone to upgrade to this version.

Highlights:

- Integration of CuDNN for 2D convolutions and pooling on supported GPUs
- Too many optimizations and new features to count
- Various fixes and improvements to scan
- Better support for GPU on Windows
- On Mac OS X, clang is used by default
- Many crash fixes
- Some bug fixes as well

6.2.2 Theano at a Glance

Theano is a Python library that lets you to define, optimize, and evaluate mathematical expressions, especially ones with multi-dimensional arrays (numpy.ndarray). Using Theano it is possible to attain speeds rivaling hand-crafted C implementations for problems involving large amounts of data. It can also surpass C on a CPU by many orders of magnitude by taking advantage of recent GPUs.

Theano combines aspects of a computer algebra system (CAS) with aspects of an optimizing compiler. It can also generate customized C code for many mathematical operations. This combination of CAS with optimizing compilation is particularly useful for tasks in which complicated mathematical expressions are evaluated repeatedly and evaluation speed is critical. For situations where many different expressions are each evaluated once Theano can minimize the amount of compilation/analysis overhead, but still provide symbolic features such as automatic differentiation.

Theano's compiler applies many optimizations of varying complexity to these symbolic expressions. These optimizations include, but are not limited to:

- use of GPU for computations
- constant folding
- merging of similar subgraphs, to avoid redundant calculation
- arithmetic simplification (e.g. `x*y/x -> y`, `--x -> x`)
- inserting efficient BLAS operations (e.g. `GEMM`) in a variety of contexts
- using memory aliasing to avoid calculation
- using inplace operations wherever it does not interfere with aliasing
- loop fusion for elementwise sub-expressions
- improvements to numerical stability (e.g. $\log(1+\exp(x))$ and $\log(\sum_i \exp(x[i]))$)
- for a complete list, see *Optimizations*

Theano was written at the LISA lab to support rapid development of efficient machine learning algorithms. Theano is named after the Greek mathematician, who may have been Pythagoras' wife. Theano is released under a BSD license (*link*).

Sneak peek

Here is an example of how to use Theano. It doesn't show off many of Theano's features, but it illustrates concretely what Theano is.

```
import theano
from theano import tensor

# declare two symbolic floating-point scalars
a = tensor.dscalar()
b = tensor.dscalar()

# create a simple expression
c = a + b

# convert the expression into a callable object that takes (a,b)
# values as input and computes a value for c
f = theano.function([a,b], c)

# bind 1.5 to 'a', 2.5 to 'b', and evaluate 'c'
assert 4.0 == f(1.5, 2.5)
```

Theano is not a programming language in the normal sense because you write a program in Python that builds expressions for Theano. Still it is like a programming language in the sense that you have to

- declare variables (`a,b`) and give their types
- build expressions for how to put those variables together

- compile expression graphs to functions in order to use them for computation.

It is good to think of `theano.function` as the interface to a compiler which builds a callable object from a purely symbolic graph. One of Theano's most important features is that `theano.function` can optimize a graph and even compile some or all of it into native machine instructions.

What does it do that they don't?

Theano is a Python library and optimizing compiler for manipulating and evaluating expressions, especially matrix-valued ones. Manipulation of matrices is typically done using the numpy package, so what does Theano do that Python and numpy do not?

- *execution speed optimizations*: Theano can use *g++* or *nvcc* to compile parts your expression graph into CPU or GPU instructions, which run much faster than pure Python.
- *symbolic differentiation*: Theano can automatically build symbolic graphs for computing gradients.
- *stability optimizations*: Theano can recognize [some] numerically unstable expressions and compute them with more stable algorithms.

The closest Python package to Theano is sympy. Theano focuses more on tensor expressions than Sympy, and has more machinery for compilation. Sympy has more sophisticated algebra rules and can handle a wider variety of mathematical operations (such as series, limits, and integrals).

If numpy is to be compared to MATLAB and sympy to Mathematica, Theano is a sort of hybrid of the two which tries to combine the best of both worlds.

Getting started

Installing Theano Instructions to download and install Theano on your system.

Tutorial Getting started with Theano's basic features. Go here if you are new!

Library Documentation Details of what Theano provides. It is recommended to go through the *Tutorial* first though.

A PDF version of the online documentation may be found here.

Theano Vision

This is the vision we have for Theano. This is give people an idea of what to expect in the future of Theano, but we can't promise to implement all of it. This should also help you to understand where Theano fits in relation to other computational tools.

- Support tensor and sparse operations
- Support linear algebra operations
- **Graph Transformations**
  - Differentiation/higher order differentiation
  - 'R' and 'L' differential operators

  - Speed/memory optimizations
  - Numerical stability optimizations
- Can use many compiled languages, instructions sets: C/C++, CUDA, OpenCL, PTX, CAL, AVX, ...
- Lazy evaluation
- Loop
- Parallel execution (SIMD, multi-core, multi-node on cluster, multi-node distributed)
- Support all NumPy/basic SciPy functionality
- Easy wrapping of library functions in Theano

Note: There is no short term plan to support multi-node computation.

Theano Vision State

Here is the state of that vision as of December 3th, 2013 (after Theano release 0.6):

- We support tensors using the *numpy.ndarray* object and we support many operations on them.
- We support sparse types by using the *scipy.{csc,csr,bsr}_matrix* object and support some operations on them.
- We have started implementing/wrapping more advanced linear algebra operations.
- We have many graph transformations that cover the 4 categories listed above.
- We can improve the graph transformation with better storage optimization and instruction selection.
  - Similar to auto-tuning during the optimization phase, but this doesn't apply to only 1 op.
  - Example of use: Determine if we should move computation to the GPU or not depending on the input size.
  - Possible implementation note: allow Theano Variable in the fgraph to have more than 1 owner.
- We have a CUDA backend for tensors of type *float32* only.
- Efforts have begun towards a generic GPU ndarray (GPU tensor) (started in the libgpuarray project)
  - Move GPU backend outside of Theano.
  - Will provide better support for GPU on Windows and support an OpenCL backend on CPU.
- Loops work, but not all related optimizations are currently done.
- The cvm linker allows lazy evaluation. It is the current default linker.
  - How to have *DebugMode* check it? Right now, DebugMode checks the computation non-lazily.
- SIMD parallelism on the CPU comes from the compiler.
- Multi-core parallelism support limited. If the external BLAS implementation supports it, many dot are parallelized via gemm, gemv and ger. Also, element-wise operation are supported. See *Multi cores support in Theano*.

- No multi-node support.
- Most, but not all NumPy functions/aliases are implemented. * https://github.com/Theano/Theano/issues/1080
- Wrapping an existing Python function in easy and documented.
- We know how to separate the shared variable memory storage location from its object type (tensor, sparse, dtype, broadcast flags), but we need to do it.

Contact us

Discussion about Theano takes place in the theano-dev and theano-users mailing lists. People interested in development of Theano should check the former, while the latter is reserved for issues that concern the end users.

Questions, comments, praise, criticism as well as bug reports should be submitted to these mailing lists.

We welcome all kinds of contributions. If you have any questions regarding how to extend Theano, please feel free to ask on the theano-dev mailing list.

6.2.3 Installing Theano

Note: If you are a member of LISA Labo, have a look at *LISA Labo specific instructions* for lab-specific installation instructions.

Requirements

In order to use Theano, the following libraries and software will need to be installed (MacOS and Windows users should refer to platform-specific instructions below for detailed installation steps):

Linux, Mac OS X or Windows operating system We develop mainly on 64-bit Linux machines. other architectures are not well-tested.

Python >= 2.6 The development package (`python-dev` or `python-devel` on most Linux distributions) is recommended (see just below). Python 2.4 was supported up to and including the release 0.6. Python 3 is supported via 2to3 only, starting from 3.3.

`g++`, `python-dev` Not technically required but *highly* recommended, in order to compile generated C code. Theano *can* fall back on a NumPy-based Python execution model, but a C compiler allows for vastly faster execution. g++ >= 4.2 (for openmp that is currently always used) more recent version recommended!

NumPy >= 1.6.2 Earlier versions could work, but we don't test it.

SciPy >= 0.11 Only currently required for sparse matrix and special functions support, but *highly* recommended. SciPy >=0.8 could work, but earlier versions have known bugs with sparse matrices.

A BLAS installation (with Level 3 functionality) Including the development headers (`-dev`, `-devel`, depending on your Linux distribution). Mac OS X comes with the Accelerate framework built in, and various options exist for Windows (see below).

The following libraries and software are optional:

nose Recommended, to run Theano's test-suite.

Sphinx >= 0.5.1, pygments For building the documentation. LaTeX and dvipng are also necessary for math to show up as images.

Git To download bleeding-edge versions of Theano.

pydot To be able to make picture of Theano computation graph.

NVIDIA CUDA drivers and SDK Required for GPU code generation/execution on NVIDIA gpus

libgpuarray Required for GPU/CPU code generation on CUDA and OpenCL devices (see: *GpuArray Backend*.)

note OpenCL support is still minimal for now.

Linux

CentOS 6

install_centos6 provides instructions on how to install Theano on CentOS 6, written by the Theano developers. It covers how to install Theano (for CPU-based computation only) with the distribution-packaged ATLAS, a free fast implementation of BLAS.

Ubuntu

install_ubuntu provides instructions on how to install Theano on Ubuntu. It covers how to install Theano with the distribution-packaged OpenBlas or ATLAS. Both are free fast implementation of BLAS.

Alternative installation on Gentoo

Brian Vandenberg emailed installation instructions on Gentoo, focusing on how to install the appropriate dependencies.

Nicolas Pinto provides ebuild scripts.

Alternative installation on Mandriva 2010.2

A contributor made rpm package for Mandriva 2010.2 of Theano 0.3.1.

Basic user install instructions

The easiest way to obtain the released version of Theano is from PyPI using pip (a replacement for easy_install provided by setuptools/distribute) by typing

```
pip install Theano
```

You may need to add `sudo` before this command to install into your system's `site-packages` directory. If you do not have administrator access to your machine, you can install Theano locally (to ~/.local) using

```
pip install Theano --user
```

Alternatively you can use virtualenv to create an isolated `site-packages` directory; see the virtualenv documentation for details.

Note: Theano *can* be installed with easy_install, however we recommend pip. `pip` offers many benefits over `easy_install` such as more intelligent dependency management, better error messages and a `pip uninstall` command for easily removing packages.

If you do not have `pip` installed but do have `easy_install`, you can get `pip` by simply typing `easy_install pip`.

Updating Theano

The following command will update only Theano:

```
sudo pip install --upgrade --no-deps theano
```

The following command will update Theano and Numpy/Scipy (warning bellow):

```
sudo pip install --upgrade theano
```

If you installed NumPy/SciPy with yum/apt-get, updating NumPy/SciPy with pip/easy_install is not always a good idea. This can make Theano crash due to problems with BLAS (but see below). The versions of NumPy/SciPy in the distribution are sometimes linked against faster versions of BLAS. Installing NumPy/SciPy with yum/apt-get/pip/easy_install won't install the development package needed to recompile it with the fast version. This mean that if you don't install the development packages manually, when you recompile the updated NumPy/SciPy, it will compile with the slower version. This results in a slower Theano as well. To fix the crash, you can clear the Theano cache like this:

```
theano-cache clear
```

Bleeding-edge install instructions

If you are a developer of Theano, then check out the *Developer Start Guide*.

If you want the bleeding-edge without developing the code you can use pip for this with the command line below. Note that it will also try to install Theano's dependencies (like NumPy and SciPy), but not upgrade

them. If you wish to upgrade them, remove the `--no-deps` switch to it, but go see a previous warning before doing this.

```
pip install --upgrade --no-deps git+git://github.com/Theano/Theano.git
```

or (if you want to install it for the current user only):

```
pip install --upgrade --no-deps git+git://github.com/Theano/Theano.git --user
```

The following are general instructions that will set you up with the bleeding-edge version of Theano and allow you to hack it. First, get the code using Git:

```
git clone git://github.com/Theano/Theano.git
```

From here, the easiest way to get started is (this requires setuptools or distribute to be installed):

```
cd Theano
python setup.py develop
```

Note: "python setup.py develop ..." does not work on Python 3 as it does not call the converter from Python 2 code to Python 3 code.

This will install a `.pth` file in your `site-packages` directory that tells Python where to look for your Theano installation (i.e. in the directory your just checked out of Github). Using `develop` mode is preferable to `install` as any modifications you make in the checkout directory (or changes you pull with Git) will be automatically reflected in the "installed" version without re-running `python setup.py install`.

If you do not have permission to modify your `site-packages` directory you can specify an alternative installation prefix using

```
python setup.py develop --prefix=~/.local
```

A common choice is `~/.local` which is automatically searched for Python >= 2.6; for earlier Python versions and other installation prefixes, the prefix specified must contain `lib/pythonA.B/site-packages`, where `A.B` is e.g. 2.5, and this `site-packages` directory must be listed in `PYTHONPATH`.

An alternative, perhaps simpler way of creating and using an isolated `site-packages` is to use virtualenv; see the virtualenv documentation for details. If you find yourself using virtualenv frequently you may find the virtualenvwrapper package useful for switching between them.

Configuring `PYTHONPATH` If `import theano` does not work in Python, you may need modify the environment variable `PYTHONPATH` accordingly. In bash, you may do this:

```
export PYTHONPATH=<new location to add>:$PYTHONPATH
```

In csh:

```
setenv PYTHONPATH <new location to add>:$PYTHONPATH
```

To make this change stick you will usually need to add the above command to your shell's startup script, i.e. `~/.bashrc` or `~/.cshrc`. Consult your shell's documentation for details.

Updating To update your library to the latest revision, change directory (`cd`) to your `Theano` folder and execute the following command:

```
git pull
```

You should update frequently, bugs are fixed on a very regular basis.

Testing your installation

Once you have installed Theano, you should run the test suite. At a Python (or IPython) interpreter,

```
>>> import theano
>>> theano.test()
```

You can also run them in-place from the Git checkout directory by typing

```
theano-nose
```

You should be able to execute it if you followed the instructions above. If `theano-nose` is not found by your shell, you will need to add `Theano/bin` to your `PATH` environment variable.

Note: In Theano versions <= 0.5, `theano-nose` was not included. If you are working with such a version, you can call `nosetests` instead of `theano-nose`. In that case, some tests will fail by raising the KnownFailureTest Exception, and will be considered as errors, but they are nothing to worry about.

Note: The tests should be run with the configuration option `device` set to `cpu` (default). If you need to change this value, you can do that by setting the `THEANO_FLAGS` environment variable, by prefixing the `theano-nose` command with `THEANO_FLAGS=device=cpu`. If you have a GPU, it will automatically be used to run GPU-related tests.

If you want GPU-related tests to run on a specific GPU device, and not the default one, you should use `init_gpu_device`. For instance: `THEANO_FLAGS=device=cpu,init_gpu_device=gpu1`.

See *config – Theano Configuration* for more information on how to change these configuration options.

All tests should pass (skipped tests and known failures are normal). If some test fails on your machine, you are encouraged to tell us what went wrong on the `theano-users@googlegroups.com` mailing list.

Troubleshooting: Make sure you have a BLAS library

There are many ways to configure BLAS for Theano. This is done with the Theano flags `blas.ldflags` (*config – Theano Configuration*). The default is to use the BLAS installation information in NumPy, accessible via `numpy.distutils.__config__.show()`. You can tell theano to use a different version of BLAS, in case you did not compile NumPy with a fast BLAS or if NumPy was compiled with a static library of BLAS (the latter is not supported in Theano).

The short way to configure the Theano flags `blas.ldflags` is by setting the environment variable `THEANO_FLAGS` to `blas.ldflags=XXX` (in bash `export THEANO_FLAGS=blas.ldflags=XXX`)

The `${HOME}/.theanorc` file is the simplest way to set a relatively permanent option like this one. Add a `[blas]` section with an `ldflags` entry like this:

```
# other stuff can go here
[blas]
ldflags = -lf77blas -latlas -lgfortran #put your flags here

# other stuff can go here
```

For more information on the formatting of `~/.theanorc` and the configuration options that you can put there, see *config – Theano Configuration*.

Here are some different way to configure BLAS:

0) Do nothing and use the default config, which is to link against the same BLAS against which NumPy was built. This does not work in the case NumPy was compiled with a static library (e.g. ATLAS is compiled by default only as a static library).

1) Disable the usage of BLAS and fall back on NumPy for dot products. To do this, set the value of `blas.ldflags` as the empty string (ex: `export THEANO_FLAGS=blas.ldflags=`). Depending on the kind of matrix operations your Theano code performs, this might slow some things down (vs. linking with BLAS directly).

2) You can install the default (reference) version of BLAS if the NumPy version (against which Theano links) does not work. If you have root or sudo access in fedora you can do `sudo yum install blas blas-devel`. Under Ubuntu/Debian `sudo apt-get install libblas-dev`. Then use the Theano flags `blas.ldflags=-lblas`. Note that the default version of blas is not optimized. Using an optimized version can give up to 10x speedups in the BLAS functions that we use.

3) Install the ATLAS library. ATLAS is an open source optimized version of BLAS. You can install a precompiled version on most OSes, but if you're willing to invest the time, you can compile it to have a faster version (we have seen speed-ups of up to 3x, especially on more recent computers, against the precompiled one). On Fedora, `sudo yum install atlas-devel`. Under Ubuntu, `sudo apt-get install libatlas-base-dev libatlas-base` or `libatlas3gf-sse2` if your CPU supports SSE2 instructions. Then set the Theano flags `blas.ldflags` to `-lf77blas -latlas -lgfortran`. Note that these flags are sometimes OS-dependent.

4) Use a faster version like MKL, GOTO, ... You are on your own to install it. See the doc of that software and set the Theano flags `blas.ldflags` correctly (for example, for MKL this might be `-lmkl -lguide -lpthread` or `-lmkl_intel_lp64 -lmkl_intel_thread -lmkl_core -lguide -liomp5 -lmkl_mc -lpthread`).

Note: Make sure your BLAS libraries are available as dynamically-loadable libraries. ATLAS is often installed only as a static library. Theano is not able to use this static library. Your ATLAS installation might need to be modified to provide dynamically loadable libraries. (On Linux this typically means a library whose name ends with .so. On Windows this will be a .dll, and on OS-X it might be either a .dylib or a .so.)

This might be just a problem with the way Theano passes compilation arguments to g++, but the problem is not fixed yet.

Note: If you have problems linking with MKL, Intel Line Advisor and the MKL User Guide can help you find the correct flags to use.

Using the GPU

The first thing you'll need for Theano to use your GPU is Nvidia's GPU-programming toolchain. You should install at least the CUDA driver and the CUDA Toolkit, as described here. The CUDA Toolkit installs a folder on your computer with subfolders *bin*, *lib*, *include*, and some more too. (Sanity check: The *bin* subfolder should contain an *nvcc* program which is the compiler for GPU code.) This folder is called the *cuda root* directory. You must also add the 'lib' subdirectory (and/or 'lib64' subdirectory if you have a 64-bit Linux computer) to your `$LD_LIBRARY_PATH` environment variable.

You must then tell Theano where the CUDA root folder is, and there are three ways to do it. Any one of them is enough.

- Define a $CUDA_ROOT environment variable to equal the cuda root directory, as in `CUDA_ROOT=/path/to/cuda/root`, or
- add a `cuda.root` flag to `THEANO_FLAGS`, as in `THEANO_FLAGS='cuda.root=/path/to/cuda/root'`, or
- add a [cuda] section to your .theanorc file containing the option `root = /path/to/cuda/root`.

Note: On Debian, you can ask the software package manager to install it for you. We have a user report that this works for Debian Wheezy (7.0). When you install it this way, you won't always have the latest version, but we were told that it gets updated regularly. One big advantage is that it will be updated automatically. You can try the `sudo apt-get install nvidia-cuda-toolkit` command to install it.

Ubuntu instructions.

Once that is done, the only thing left is to change the `device` option to name the GPU device in your computer, and set the default floating point computations to float32. For example: `THEANO_FLAGS='cuda.root=/path/to/cuda/root,device=gpu,floatX=float32'`. You can also set these options in the .theanorc file's `[global]` section:

```
[global]
device = gpu
floatX = float32
```

Note that:

- If your computer has multiple GPUs and you use 'device=gpu', the driver selects the one to use (usually gpu0).
- You can use the program nvida-smi to change this policy.
- You can choose one specific GPU by specifying 'device=gpuX', with X the the corresponding GPU index (0, 1, 2, ...)

- By default, when `device` indicates preference for GPU computations, Theano will fall back to the CPU if there is a problem with the GPU. You can use the flag 'force_device=True' to instead raise an error when Theano cannot use the GPU.

Once your setup is complete, head to *Using the GPU* to find how to verify everything is working properly.

Mac OS

There are various ways to install Theano dependencies on a Mac. Here we describe the process in detail with Canopy, Anaconda, Homebrew or MacPorts but if you did it differently and it worked, please let us know the details on the theano-users mailing-list, so that we can add alternate instructions here.

In academia: Enthought Canopy

If you are working in academia, the easiest way to install most of the dependencies is to install Canopy. If you are affiliated with a university (as student or employee), you can download the installer for free.

The Canopy installation includes in particular Python (and the development headers), NumPy, SciPy, nose, sphinx, pip, pydot (but *not* Graphviz, which is necessary for it to work) and the MKL implementation of blas.

To install the latest Theano release execute this in a terminal:

```
$ pip install Theano
```

If you want the bleeding edge version execute this command instead:

```
$ pip install --upgrade --no-deps git+git://github.com/Theano/Theano.git
```

See the section install_bleeding_edge for more information on the bleeding edge version.

Then you must install the compiler. See *Installing the compiler* below.

Note: If you use version 0.6 or later of Theano, we try to automatically link with the Canopy blas version. Due to Mac OS peculiarities, this requires user intervention. We detect if the manipulation was done or not and give an error message explaining what to do in case it hasn't been done.

Anaconda

An easy way to install most of the dependencies is to install Anaconda. There is a free version available to everybody. If you install their `MKL Optimizations` product (free for academic, ~30$ otherwise) Theano will also be optimized as we will reuse the faster BLAS version automatically.

The Anaconda installation includes in particular Python (and the development headers), NumPy, SciPy, nose, sphinx, pip, and a acceptable BLAS version.

After installing Anaconda, in a terminal execute this command to install the latest Theano release:

```
$ pip install Theano
```

To install the missing Theano optional dependency (pydot):

```
$ conda install pydot
```

If you want the bleeding edge version instead execute this command:

```
$ pip install --upgrade --no-deps git+git://github.com/Theano/Theano.git
```

See the section install_bleeding_edge for more information on the bleeding edge version.

Then you must install the compiler. See *Installing the compiler* below.

Note: If you use version 0.6 or later of Theano, we try to automatically link with the python library. Due to Mac OS peculiarities, this requires user intervention. We detect if the user did the modification and if not, we tell him how to do it.

Installing the compiler

Theano officially supports only clang on OS X. This can be installed by getting XCode from the App Store and running it once to install the command-line tools.

If you still want to use g++ you can do so by setting its full path in the theano config flag *gxx*. Note that any bug reports on Mac using g++ will be ignored unless it can be reproduced with clang.

Homebrew

Install python with homebrew:

```
$ brew install python # or python3 if you prefer
```

This will install pip. Then use pip to install numpy, scipy:

```
$ pip install numpy scipy
```

If you want to use openblas instead of Accelerate, you have to install numpy and scipy with hombrew:

```
$ brew tap homebrew/python
$ brew install numpy --with-openblas
$ brew install scipy --with-openblas
```

Then install theano as usual:

```
$ pip install Theano --user
```

Or for the bleeding-edge version:

```
$ pip install --upgrade --no-deps git+git://github.com/Theano/Theano.git
```

MacPorts

Using MacPorts to install all required Theano dependencies is easy, but be aware that it will take a long time (a few hours) to build and install everything.

- MacPorts requires installing XCode first (which can be found in the Mac App Store), if you do not have it already. If you can't install it from the App Store, look in your MacOS X installation DVD for an old version. Then update your Mac to update XCode.
- Download and install MacPorts, then ensure its package list is up-to-date with `sudo port selfupdate`.
- Then, in order to install one or more of the required libraries, use `port install`, e.g. as follows:

  ```
  $ sudo port install py27-numpy +atlas py27-scipy +atlas py27-pip
  ```

  This will install all the required Theano dependencies. gcc will be automatically installed (since it is a SciPy dependency), but be aware that it takes a long time to compile (hours)! Having NumPy and SciPy linked with ATLAS (an optimized BLAS implementation) is not mandatory, but recommended if you care about performance.
- You might have some different versions of gcc, SciPy, NumPy, Python installed on your system, perhaps via Xcode. It is a good idea to use **either** the MacPorts version of everything **or** some other set of compatible versions (e.g. provided by Xcode or Fink). The advantages of MacPorts are the transparency with which everything can be installed and the fact that packages are updated quite frequently. The following steps describe how to make sure you are using the MacPorts version of these packages.
- In order to use the MacPorts version of Python, you will probably need to explicitly select it with `sudo port select python python27`. The reason this is necessary is because you may have an Apple-provided Python (via, for example, an Xcode installation). After performing this step, you should check that the symbolic link provided by `which python` points to the MacPorts python. For instance, on MacOS X Lion with MacPorts 2.0.3, the output of `which python` is `/opt/local/bin/python` and this symbolic link points to `/opt/local/bin/python2.7`. When executing `sudo port select python python27-apple` (which you should **not** do), the link points to `/usr/bin/python2.7`.
- Similarly, make sure that you are using the MacPorts-provided gcc: use `sudo port select gcc` to see which gcc installs you have on the system. Then execute for instance `sudo port select gcc mp-gcc44` to create a symlink that points to the correct (MacPorts) gcc (version 4.4 in this case).
- At this point, if you have not done so already, it may be a good idea to close and restart your terminal, to make sure all configuration changes are properly taken into account.
- Afterwards, please check that the `scipy` module that is imported in Python is the right one (and is a recent one). For instance, `import scipy` followed by `print scipy.__version__` and `print scipy.__path__` should result in a version number of at least 0.7.0 and a path that starts with `/opt/local` (the path where MacPorts installs its packages). If this is not the case, then you might have some old installation of `scipy` in your `PYTHONPATH` so you should edit `PYTHONPATH` accordingly.

- Please follow the same procedure with `numpy`.
- This is covered in the MacPorts installation process, but make sure that your `PATH` environment variable contains `/opt/local/bin` and `/opt/local/sbin` before any other paths (to ensure that the Python and gcc binaries that you installed with MacPorts are visible first).
- MacPorts does not create automatically `nosetests` and `pip` symlinks pointing to the MacPorts version, so you can add them yourself with

```
$ sudo ln -s /opt/local/bin/nosetests-2.7 /opt/local/bin/nosetests
$ sudo ln -s /opt/local/bin/pip-2.7 /opt/local/bin/pip
```

- At this point you are ready to install Theano with

```
$ sudo pip install Theano
```

  And if you are in no hurry, you can run its test-suite with

```
$ python -c "import theano; theano.test()"
```

Using the GPU

You should be able to follow the *Linux* instructions to setup CUDA, but be aware of the following caveats:

- If you want to compile the CUDA SDK code, you may need to temporarily revert back to Apple's gcc (`sudo port select gcc`) as their Makefiles are not compatible with MacPort's gcc.
- If CUDA seems unable to find a CUDA-capable GPU, you may need to manually toggle your GPU on, which can be done with gfxCardStatus.

Once your setup is complete, head to *Using the GPU* to find how to verify everything is working properly.

Troubleshooting MacOS issues

Although the above steps should be enough, running Theano on a Mac may sometimes cause unexpected crashes, typically due to multiple versions of Python or other system libraries. If you encounter such problems, you may try the following.

- You can ensure MacPorts shared libraries are given priority at run-time with `export LD_LIBRARY_PATH=/opt/local/lib:$LD_LIBRARY_PATH`. In order to do the same at compile time, you can add to your `~/.theanorc`:

```
[gcc]
cxxflags = -L/opt/local/lib
```

- An obscure `Bus error` can sometimes be caused when linking Theano-generated object files against the `framework` library in Leopard. For this reason, we have disabled linking with `-framework Python`, since on most configurations this solves the `Bus error` problem. If this default configuration causes problems with your Python/Theano installation and you think that linking with `-framework Python` might help, then either set the `THEANO_FLAGS` environment variable

with `THEANO_FLAGS=cmodule.mac_framework_link` or edit your `~/.theanorc` to contain

```
[cmodule]
mac_framework_link=True
```

- More generally, to investigate libraries issues, you can use the `otool -L` command on `.so` files found under your `~/.theano` directory. This will list shared libraries dependencies, and may help identify incompatibilities.

Please inform us if you have trouble installing and running Theano on your Mac. We would be especially interested in dependencies that we missed listing, alternate installation steps, GPU instructions, as well as tests that fail on your platform (use the `theano-users@googlegroups.com` mailing list, but note that you must first register to it, by going to theano-users).

Windows

install_windows provides step-by-step instructions on how to install Theano on 32- or 64-bit Windows systems, using freely available tools and compilers.

Editing code in Visual Studio

You will find a Visual Studio solution file (`Theano.sln`) in the root of the Theano repository. Note that this project file may not be kept up-to-date and is not officially supported by the core Theano developers: it is provided for convenience only. Also, be aware that it will not make Theano use Visual Studio to compile C files: it is only meant to provide an easy way to edit Theano code within the Visual Studio editor.

Windows Installation References

1. http://stackoverflow.com/questions/9047072/windows-python-version-and-vc-redistributable-version
2. http://stackoverflow.com/questions/1865069/how-to-compile-a-64-bit-application-using-visual-c-2010-express
3. http://blog.victorjabur.com/2011/06/05/compiling-python-2-7-modules-on-windows-32-and-64-using-msvc-2008-express/
4. http://stackoverflow.com/questions/126279/c99-stdint-h-header-and-ms-visual-studio
5. http://stackoverflow.com/questions/11182765/how-can-i-build-my-c-extensions-with-mingw-w64-in-python
6. https://mail.python.org/pipermail/python-announce-list/2014-September/010457.html

Generating the documentation

You can read the latest HTML documentation here. You can download the latest PDF documentation here.

We recommend you look at the documentation on the website, since it will be more current than the documentation included with the package.

If you really wish to build the documentation yourself, you will need epydoc and sphinx, as described above. Issue the following command:

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

Documentation is built into `html/`. The PDF of the documentation is `html/theano.pdf`.

6.2.4 Tutorial

Let us start an interactive session (e.g. with `python` or `ipython`) and import Theano.

```
>>> from theano import *
```

Several of the symbols you will need to use are in the `tensor` subpackage of Theano. Let us import that subpackage under a handy name like `T` (the tutorials will frequently use this convention).

```
>>> import theano.tensor as T
```

If that succeeded you are ready for the tutorial, otherwise check your installation (see *Installing Theano*).

Throughout the tutorial, bear in mind that there is a *Glossary* as well as *index* and *modules* links in the upper-right corner of each page to help you out.

Python tutorial

In this documentation, we suppose that the reader knows Python. Here is a small list of Python tutorials/exercises if you need to learn it or only need a refresher:

- Python Challenge
- Dive into Python
- Google Python Class
- Enthought Python course (free for academics)

We have a tutorial on how *Python manages its memory*.

NumPy refresher

Here are some quick guides to NumPy:

- Numpy quick guide for Matlab users
- Numpy User Guide
- More detailed Numpy tutorial
- 100 NumPy exercises
- Numpy tutorial

Matrix conventions for machine learning

Rows are horizontal and columns are vertical. Every row is an example. Therefore, inputs[10,5] is a matrix of 10 examples where each example has dimension 5. If this would be the input of a neural network then the weights from the input to the first hidden layer would represent a matrix of size (5, #hid).

Consider this array:

```
>>> numpy.asarray([[1., 2], [3, 4], [5, 6]])
array([[ 1.,  2.],
       [ 3.,  4.],
       [ 5.,  6.]])
>>> numpy.asarray([[1., 2], [3, 4], [5, 6]]).shape
(3, 2)
```

This is a 3x2 matrix, i.e. there are 3 rows and 2 columns.

To access the entry in the 3rd row (row #2) and the 1st column (column #0):

```
>>> numpy.asarray([[1., 2], [3, 4], [5, 6]])[2, 0]
5.0
```

To remember this, keep in mind that we read left-to-right, top-to-bottom, so each thing that is contiguous is a row. That is, there are 3 rows and 2 columns.

Broadcasting

Numpy does *broadcasting* of arrays of different shapes during arithmetic operations. What this means in general is that the smaller array (or scalar) is *broadcasted* across the larger array so that they have compatible shapes. The example below shows an instance of *broadcastaing*:

```
>>> a = numpy.asarray([1.0, 2.0, 3.0])
>>> b = 2.0
>>> a * b
array([2., 4., 6.])
```

The smaller array `b` (actually a scalar here, which works like a 0-d array) in this case is *broadcasted* to the same size as `a` during the multiplication. This trick is often useful in simplifying how expression are written. More detail about *broadcasting* can be found in the numpy user guide.

Baby Steps - Algebra

Adding two Scalars

To get us started with Theano and get a feel of what we're working with, let's make a simple function: add two numbers together. Here is how you do it:

```
>>> import theano.tensor as T
>>> from theano import function
>>> x = T.dscalar('x')
```

```
>>> y = T.dscalar('y')
>>> z = x + y
>>> f = function([x, y], z)
```

And now that we've created our function we can use it:

```
>>> f(2, 3)
array(5.0)
>>> f(16.3, 12.1)
array(28.4)
```

Let's break this down into several steps. The first step is to define two symbols (*Variables*) representing the quantities that you want to add. Note that from now on, we will use the term *Variable* to mean "symbol" (in other words, *x*, *y*, *z* are all *Variable* objects). The output of the function *f* is a `numpy.ndarray` with zero dimensions.

If you are following along and typing into an interpreter, you may have noticed that there was a slight delay in executing the `function` instruction. Behind the scene, *f* was being compiled into C code.

Step 1

```
>>> x = T.dscalar('x')
>>> y = T.dscalar('y')
```

In Theano, all symbols must be typed. In particular, `T.dscalar` is the type we assign to "0-dimensional arrays (*scalar*) of doubles (*d*)". It is a Theano *Type*.

`dscalar` is not a class. Therefore, neither *x* nor *y* are actually instances of `dscalar`. They are instances of `TensorVariable`. *x* and *y* are, however, assigned the theano Type `dscalar` in their `type` field, as you can see here:

```
>>> type(x)
<class 'theano.tensor.basic.TensorVariable'>
>>> x.type
TensorType(float64, scalar)
>>> T.dscalar
TensorType(float64, scalar)
>>> x.type is T.dscalar
True
```

By calling `T.dscalar` with a string argument, you create a *Variable* representing a floating-point scalar quantity with the given name. If you provide no argument, the symbol will be unnamed. Names are not required, but they can help debugging.

More will be said in a moment regarding Theano's inner structure. You could also learn more by looking into *Graph Structures*.

Step 2

The second step is to combine *x* and *y* into their sum *z*:

```
>>> z = x + y
```

z is yet another *Variable* which represents the addition of *x* and *y*. You can use the *pp* function to pretty-print out the computation associated to *z*.

```
>>> from theano import pp
>>> print pp(z)
(x + y)
```

Step 3

The last step is to create a function taking *x* and *y* as inputs and giving *z* as output:

```
>>> f = function([x, y], z)
```

The first argument to `function` is a list of Variables that will be provided as inputs to the function. The second argument is a single Variable *or* a list of Variables. For either case, the second argument is what we want to see as output when we apply the function. *f* may then be used like a normal Python function.

Note: As a shortcut, you can skip step 3, and just use a variable's `eval()` method. The `eval()` method is not as flexible as `function()` but it can do everything we've covered in the tutorial so far. It has the added benefit of not requiring you to import `function()` . Here is how `eval()` works:

```
>>> import theano.tensor as T
>>> x = T.dscalar('x')
>>> y = T.dscalar('y')
>>> z = x + y
>>> z.eval({x : 16.3, y : 12.1})
array(28.4)
```

We passed `eval()` a dictionary mapping symbolic theano variables to the values to substitute for them, and it returned the numerical value of the expression.

`eval()` will be slow the first time you call it on a variable – it needs to call `function()` to compile the expression behind the scenes. Subsequent calls to `eval()` on that same variable will be fast, because the variable caches the compiled function.

Adding two Matrices

You might already have guessed how to do this. Indeed, the only change from the previous example is that you need to instantiate *x* and *y* using the matrix Types:

```
>>> x = T.dmatrix('x')
>>> y = T.dmatrix('y')
>>> z = x + y
>>> f = function([x, y], z)
```

`dmatrix` is the Type for matrices of doubles. Then we can use our new function on 2D arrays:

```
>>> f([[1, 2], [3, 4]], [[10, 20], [30, 40]])
array([[ 11.,  22.],
       [ 33.,  44.]])
```

The variable is a NumPy array. We can also use NumPy arrays directly as inputs:

```
>>> import numpy
>>> f(numpy.array([[1, 2], [3, 4]]), numpy.array([[10, 20], [30, 40]]))
array([[ 11.,  22.],
       [ 33.,  44.]])
```

It is possible to add scalars to matrices, vectors to matrices, scalars to vectors, etc. The behavior of these operations is defined by *broadcasting*.

The following types are available:

- **byte**: `bscalar, bvector, bmatrix, brow, bcol, btensor3, btensor4`
- **16-bit integers**: `wscalar, wvector, wmatrix, wrow, wcol, wtensor3, wtensor4`
- **32-bit integers**: `iscalar, ivector, imatrix, irow, icol, itensor3, itensor4`
- **64-bit integers**: `lscalar, lvector, lmatrix, lrow, lcol, ltensor3, ltensor4`
- **float**: `fscalar, fvector, fmatrix, frow, fcol, ftensor3, ftensor4`
- **double**: `dscalar, dvector, dmatrix, drow, dcol, dtensor3, dtensor4`
- **complex**: `cscalar, cvector, cmatrix, crow, ccol, ctensor3, ctensor4`

The previous list is not exhaustive and a guide to all types compatible with NumPy arrays may be found here: *tensor creation*.

Note: You, the user—not the system architecture—have to choose whether your program will use 32- or 64-bit integers (`i` prefix vs. the `l` prefix) and floats (`f` prefix vs. the `d` prefix).

Exercise

```
import theano
a = theano.tensor.vector() # declare variable
out = a + a ** 10               # build symbolic expression
f = theano.function([a], out)   # compile function
print f([0, 1, 2])  # prints 'array([0, 2, 1026])'
```

Modify and execute this code to compute this expression: a ** 2 + b ** 2 + 2 * a * b.

`Solution`

More Examples

At this point it would be wise to begin familiarizing yourself more systematically with Theano's fundamental objects and operations by browsing this section of the library: *Basic Tensor Functionality*.

As the tutorial unfolds, you should also gradually acquaint yourself with the other relevant areas of the library and with the relevant subjects of the documentation entrance page.

Logistic Function

Here's another straightforward example, though a bit more elaborate than adding two numbers together. Let's say that you want to compute the logistic curve, which is given by:

$$s(x) = \frac{1}{1 + e^{-x}}$$

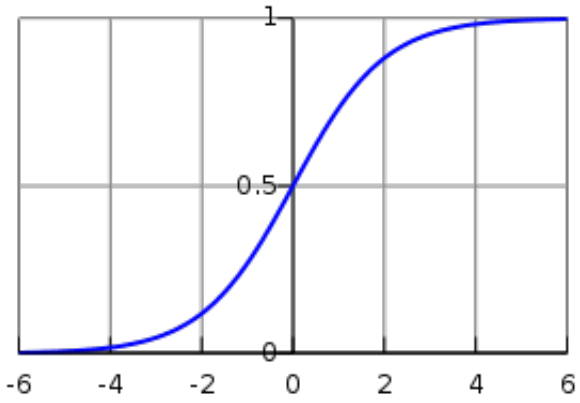


Figure 6.1: A plot of the logistic function, with x on the x-axis and s(x) on the y-axis.

You want to compute the function *elementwise* on matrices of doubles, which means that you want to apply this function to each individual element of the matrix.

Well, what you do is this:

```
>>> x = T.dmatrix('x')
>>> s = 1 / (1 + T.exp(-x))
>>> logistic = function([x], s)
>>> logistic([[0, 1], [-1, -2]])
array([[ 0.5        ,  0.73105858],
       [ 0.26894142,  0.11920292]])
```

The reason logistic is performed elementwise is because all of its operations—division, addition, exponentiation, and division—are themselves elementwise operations.

It is also the case that:

$$s(x) = \frac{1}{1 + e^{-x}} = \frac{1 + \tanh(x/2)}{2}$$

We can verify that this alternate form produces the same values:

```
>>> s2 = (1 + T.tanh(x / 2)) / 2
>>> logistic2 = function([x], s2)
>>> logistic2([[0, 1], [-1, -2]])
array([[ 0.5        ,  0.73105858],
       [ 0.26894142,  0.11920292]])
```

Computing More than one Thing at the Same Time

Theano supports functions with multiple outputs. For example, we can compute the *elementwise* difference, absolute difference, and squared difference between two matrices *a* and *b* at the same time:

```
>>> a, b = T.dmatrices('a', 'b')
>>> diff = a - b
>>> abs_diff = abs(diff)
>>> diff_squared = diff**2
>>> f = function([a, b], [diff, abs_diff, diff_squared])
```

Note: *dmatrices* produces as many outputs as names that you provide. It is a shortcut for allocating symbolic variables that we will often use in the tutorials.

When we use the function f, it returns the three variables (the printing was reformatted for readability):

```
>>> f([[1, 1], [1, 1]], [[0, 1], [2, 3]])
[array([[ 1.,  0.],
        [-1., -2.]]),
 array([[ 1.,  0.],
        [ 1.,  2.]]),
 array([[ 1.,  0.],
        [ 1.,  4.]])]
```

Setting a Default Value for an Argument

Let's say you want to define a function that adds two numbers, except that if you only provide one number, the other input is assumed to be one. You can do it like this:

```
>>> from theano import Param
>>> x, y = T.dscalars('x', 'y')
>>> z = x + y
>>> f = function([x, Param(y, default=1)], z)
>>> f(33)
array(34.0)
>>> f(33, 2)
array(35.0)
```

This makes use of the *Param* class which allows you to specify properties of your function's parameters with greater detail. Here we give a default value of 1 for *y* by creating a `Param` instance with its `default` field set to 1.

Inputs with default values must follow inputs without default values (like Python's functions). There can be multiple inputs with default values. These parameters can be set positionally or by name, as in standard Python:

```
>>> x, y, w = T.dscalars('x', 'y', 'w')
>>> z = (x + y) * w
>>> f = function([x, Param(y, default=1), Param(w, default=2, name='w_by_name')], z)
>>> f(33)
array(68.0)
```

```
>>> f(33, 2)
array(70.0)
>>> f(33, 0, 1)
array(33.0)
>>> f(33, w_by_name=1)
array(34.0)
>>> f(33, w_by_name=1, y=0)
array(33.0)
```

Note: `Param` does not know the name of the local variables *y* and *w* that are passed as arguments. The symbolic variable objects have name attributes (set by `dscalars` in the example above) and *these* are the names of the keyword parameters in the functions that we build. This is the mechanism at work in `Param(y, default=1)`. In the case of `Param(w, default=2, name='w_by_name')`. We override the symbolic variable's name attribute with a name to be used for this function.

You may like to see *Function* in the library for more detail.

Using Shared Variables

It is also possible to make a function with an internal state. For example, let's say we want to make an accumulator: at the beginning, the state is initialized to zero. Then, on each function call, the state is incremented by the function's argument.

First let's define the *accumulator* function. It adds its argument to the internal state, and returns the old state value.

```
>>> from theano import shared
>>> state = shared(0)
>>> inc = T.iscalar('inc')
>>> accumulator = function([inc], state, updates=[(state, state+inc)])
```

This code introduces a few new concepts. The `shared` function constructs so-called *shared variables*. These are hybrid symbolic and non-symbolic variables whose value may be shared between multiple functions. Shared variables can be used in symbolic expressions just like the objects returned by `dmatrices(...)` but they also have an internal value that defines the value taken by this symbolic variable in *all* the functions that use it. It is called a *shared* variable because its value is shared between many functions. The value can be accessed and modified by the `.get_value()` and `.set_value()` methods. We will come back to this soon.

The other new thing in this code is the `updates` parameter of `function`. `updates` must be supplied with a list of pairs of the form (shared-variable, new expression). It can also be a dictionary whose keys are shared-variables and values are the new expressions. Either way, it means "whenever this function runs, it will replace the `.value` of each shared variable with the result of the corresponding expression". Above, our accumulator replaces the `state`'s value with the sum of the state and the increment amount.

Let's try it out!

```
>>> state.get_value()
array(0)
>>> accumulator(1)
```

```
array(0)
>>> state.get_value()
array(1)
>>> accumulator(300)
array(1)
>>> state.get_value()
array(301)
```

It is possible to reset the state. Just use the `.set_value()` method:

```
>>> state.set_value(-1)
>>> accumulator(3)
array(-1)
>>> state.get_value()
array(2)
```

As we mentioned above, you can define more than one function to use the same shared variable. These functions can all update the value.

```
>>> decrementor = function([inc], state, updates=[(state, state-inc)])
>>> decrementor(2)
array(2)
>>> state.get_value()
array(0)
```

You might be wondering why the updates mechanism exists. You can always achieve a similar result by returning the new expressions, and working with them in NumPy as usual. The updates mechanism can be a syntactic convenience, but it is mainly there for efficiency. Updates to shared variables can sometimes be done more quickly using in-place algorithms (e.g. low-rank matrix updates). Also, Theano has more control over where and how shared variables are allocated, which is one of the important elements of getting good performance on the *GPU*.

It may happen that you expressed some formula using a shared variable, but you do *not* want to use its value. In this case, you can use the `givens` parameter of `function` which replaces a particular node in a graph for the purpose of one particular function.

```
>>> fn_of_state = state * 2 + inc
>>> # The type of foo must match the shared variable we are replacing
>>> # with the ''givens''
>>> foo = T.scalar(dtype=state.dtype)
>>> skip_shared = function([inc, foo], fn_of_state,
                           givens=[(state, foo)])
>>> skip_shared(1, 3)  # we're using 3 for the state, not state.value
array(7)
>>> state.get_value()  # old state still there, but we didn't use it
array(0)
```

The `givens` parameter can be used to replace any symbolic variable, not just a shared variable. You can replace constants, and expressions, in general. Be careful though, not to allow the expressions introduced by a `givens` substitution to be co-dependent, the order of substitution is not defined, so the substitutions have to work in any order.

In practice, a good way of thinking about the `givens` is as a mechanism that allows you to replace any part

of your formula with a different expression that evaluates to a tensor of same shape and dtype.

Note: Theano shared variable broadcast pattern default to False for each dimensions. Shared variable size can change over time, so we can't use the shape to find the broadcastable pattern. If you want a different pattern, just pass it as a parameter `theano.shared(..., broadcastable=(True, False))`

Using Random Numbers

Because in Theano you first express everything symbolically and afterwards compile this expression to get functions, using pseudo-random numbers is not as straightforward as it is in NumPy, though also not too complicated.

The way to think about putting randomness into Theano's computations is to put random variables in your graph. Theano will allocate a NumPy RandomStream object (a random number generator) for each such variable, and draw from it as necessary. We will call this sort of sequence of random numbers a *random stream*. *Random streams* are at their core shared variables, so the observations on shared variables hold here as well. Theanos's random objects are defined and implemented in *RandomStreams* and, at a lower level, in *RandomStreamsBase*.

Brief Example Here's a brief example. The setup code is:

```
from theano.tensor.shared_randomstreams import RandomStreams
from theano import function
srng = RandomStreams(seed=234)
rv_u = srng.uniform((2,2))
rv_n = srng.normal((2,2))
f = function([], rv_u)
g = function([], rv_n, no_default_updates=True)    #Not updating rv_n.rng
nearly_zeros = function([], rv_u + rv_u - 2 * rv_u)
```

Here, 'rv_u' represents a random stream of 2x2 matrices of draws from a uniform distribution. Likewise, 'rv_n' represents a random stream of 2x2 matrices of draws from a normal distribution. The distributions that are implemented are defined in `RandomStreams` and, at a lower level, in *raw_random*. They only work on CPU. See Other Implementations for GPU version.

Now let's use these objects. If we call f(), we get random uniform numbers. The internal state of the random number generator is automatically updated, so we get different random numbers every time.

```
>>> f_val0 = f()
>>> f_val1 = f()  #different numbers from f_val0
```

When we add the extra argument `no_default_updates=True` to `function` (as in *g*), then the random number generator state is not affected by calling the returned function. So, for example, calling *g* multiple times will return the same numbers.

```
>>> g_val0 = g()  # different numbers from f_val0 and f_val1
>>> g_val1 = g()  # same numbers as g_val0!
```

An important remark is that a random variable is drawn at most once during any single function execution. So the *nearly_zeros* function is guaranteed to return approximately 0 (except for rounding error) even though the *rv_u* random variable appears three times in the output expression.

```
>>> nearly_zeros = function([], rv_u + rv_u - 2 * rv_u)
```

Seeding Streams Random variables can be seeded individually or collectively.

You can seed just one random variable by seeding or assigning to the `.rng` attribute, using `.rng.set_value()`.

```
>>> rng_val = rv_u.rng.get_value(borrow=True)   # Get the rng for rv_u
>>> rng_val.seed(89234)                         # seeds the generator
>>> rv_u.rng.set_value(rng_val, borrow=True)    # Assign back seeded rng
```

You can also seed *all* of the random variables allocated by a `RandomStreams` object by that object's `seed` method. This seed will be used to seed a temporary random number generator, that will in turn generate seeds for each of the random variables.

```
>>> srng.seed(902340)  # seeds rv_u and rv_n with different seeds each
```

Sharing Streams Between Functions As usual for shared variables, the random number generators used for random variables are common between functions. So our *nearly_zeros* function will update the state of the generators used in function *f* above.

For example:

```
>>> state_after_v0 = rv_u.rng.get_value().get_state()
>>> nearly_zeros()       # this affects rv_u's generator
>>> v1 = f()
>>> rng = rv_u.rng.get_value(borrow=True)
>>> rng.set_state(state_after_v0)
>>> rv_u.rng.set_value(rng, borrow=True)
>>> v2 = f()             # v2 != v1
>>> v3 = f()             # v3 == v1
```

Copying Random State Between Theano Graphs In some use cases, a user might want to transfer the "state" of all random number generators associated with a given theano graph (e.g. g1, with compiled function f1 below) to a second graph (e.g. g2, with function f2). This might arise for example if you are trying to initialize the state of a model, from the parameters of a pickled version of a previous model. For `theano.tensor.shared_randomstreams.RandomStreams` and `theano.sandbox.rng_mrg.MRG_RandomStreams` this can be achieved by copying elements of the *state_updates* parameter.

Each time a random variable is drawn from a RandomStreams object, a tuple is added to the *state_updates* list. The first element is a shared variable, which represents the state of the random number generator associated with this *particular* variable, while the second represents the theano graph corresponding to the random number generation process (i.e. RandomFunction{uniform}.0).

An example of how "random states" can be transferred from one theano function to another is shown below.

```
import theano
import numpy
import theano.tensor as T
from theano.sandbox.rng_mrg import MRG_RandomStreams
from theano.tensor.shared_randomstreams import RandomStreams

class Graph():
    def __init__(self, seed=123):
        self.rng = RandomStreams(seed)
        self.y = self.rng.uniform(size=(1,))

g1 = Graph(seed=123)
f1 = theano.function([], g1.y)

g2 = Graph(seed=987)
f2 = theano.function([], g2.y)

print 'By default, the two functions are out of sync.'
print 'f1() returns ', f1()
print 'f2() returns ', f2()

def copy_random_state(g1, g2):
    if isinstance(g1.rng, MRG_RandomStreams):
        g2.rng.rstate = g1.rng.rstate
    for (su1, su2) in zip(g1.rng.state_updates, g2.rng.state_updates):
        su2[0].set_value(su1[0].get_value())

print 'We now copy the state of the theano random number generators.'
copy_random_state(g1, g2)
print 'f1() returns ', f1()
print 'f2() returns ', f2()
```

This gives the following output:

```
# By default, the two functions are out of sync.
f1() returns  [ 0.72803009]
f2() returns  [ 0.55056769]
# We now copy the state of the theano random number generators.
f1() returns  [ 0.59044123]
f2() returns  [ 0.59044123]
```

Other Random Distributions There are *other distributions implemented*.

Other Implementations There are 2 other implementations based on *MRG31k3p* and `CURAND`. The RandomStream only work on the CPU, MRG31k3p work on the CPU and GPU. CURAND only work on the GPU.

Note: To use you the MRG version easily, you can just change the import to:

```
from theano.sandbox.rng_mrg import MRG_RandomStreams as RandomStreams
```

A Real Example: Logistic Regression

The preceding elements are featured in this more realistic example. It will be used repeatedly.

```
import numpy
import theano
import theano.tensor as T
rng = numpy.random

N = 400
feats = 784
D = (rng.randn(N, feats), rng.randint(size=N, low=0, high=2))
training_steps = 10000

# Declare Theano symbolic variables
x = T.matrix("x")
y = T.vector("y")
w = theano.shared(rng.randn(feats), name="w")
b = theano.shared(0., name="b")
print "Initial model:"
print w.get_value(), b.get_value()

# Construct Theano expression graph
p_1 = 1 / (1 + T.exp(-T.dot(x, w) - b))   # Probability that target = 1
prediction = p_1 > 0.5                    # The prediction thresholded
xent = -y * T.log(p_1) - (1-y) * T.log(1-p_1) # Cross-entropy loss function
cost = xent.mean() + 0.01 * (w ** 2).sum()# The cost to minimize
gw, gb = T.grad(cost, [w, b])             # Compute the gradient of the cost
                                          # (we shall return to this in a
                                          # following section of this tutorial)

# Compile
train = theano.function(
          inputs=[x,y],
          outputs=[prediction, xent],
          updates=((w, w - 0.1 * gw), (b, b - 0.1 * gb)))
predict = theano.function(inputs=[x], outputs=prediction)

# Train
for i in range(training_steps):
    pred, err = train(D[0], D[1])

print "Final model:"
print w.get_value(), b.get_value()
print "target values for D:", D[1]
print "prediction on D:", predict(D[0])
```

Graph Structures

Theano Graphs

Debugging or profiling code written in Theano is not that simple if you do not know what goes on under the hood. This chapter is meant to introduce you to a required minimum of the inner workings of Theano. For more detail see *Extending Theano*.

The first step in writing Theano code is to write down all mathematical relations using symbolic placeholders (**variables**). When writing down these expressions you use operations like `+`, `-`, `**`, `sum()`, `tanh()`. All these are represented internally as **ops**. An *op* represents a certain computation on some type of inputs producing some type of output. You can see it as a *function definition* in most programming languages.

Theano builds internally a graph structure composed of interconnected **variable** nodes, **op** nodes and **apply** nodes. An *apply* node represents the application of an *op* to some *variables*. It is important to draw the difference between the definition of a computation represented by an *op* and its application to some actual data which is represented by the *apply* node. For more detail about these building blocks refer to *Variable*, *Op*, *Apply*. Here is an example of a graph:

Code

```
x = T.dmatrix('x')
y = T.dmatrix('y')
z = x + y
```

Diagram

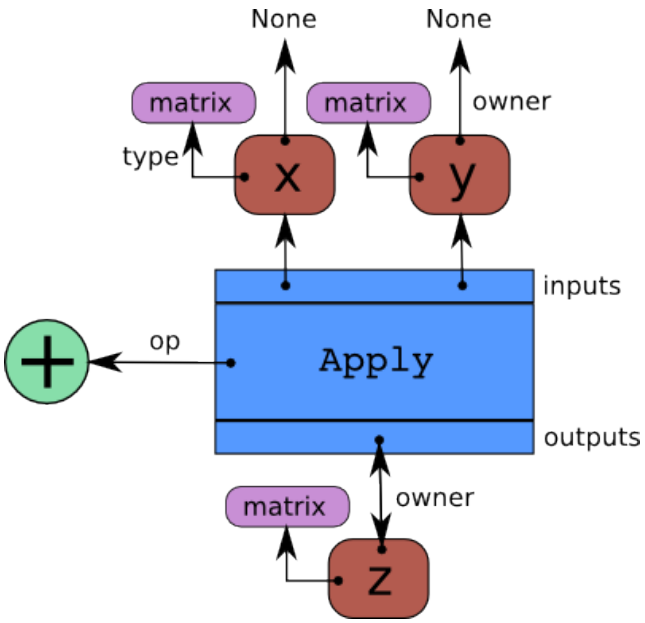


Figure 6.2: Interaction between instances of Apply (blue), Variable (red), Op (green), and Type (purple).

Arrows in this figure represent references to the Python objects pointed at. The blue box is an *Apply* node. Red boxes are *Variable* nodes. Green circles are *Ops*. Purple boxes are *Types*.

The graph can be traversed starting from outputs (the result of some computation) down to its inputs using the owner field. Take for example the following code:

```
>>> import theano
>>> x = theano.tensor.dmatrix('x')
>>> y = x * 2.
```

If you enter `type(y.owner)` you get `<class 'theano.gof.graph.Apply'>`, which is the apply node that connects the op and the inputs to get this output. You can now print the name of the op that is applied to get *y*:

```
>>> y.owner.op.name
'Elemwise{mul,no_inplace}'
```

Hence, an elementwise multiplication is used to compute *y*. This multiplication is done between the inputs:

```
>>> len(y.owner.inputs)
2
>>> y.owner.inputs[0]
x
>>> y.owner.inputs[1]
DimShuffle{x,x}.0
```

Note that the second input is not 2 as we would have expected. This is because 2 was first *broadcasted* to a matrix of same shape as *x*. This is done by using the op `DimShuffle` :

```
>>> type(y.owner.inputs[1])
<class 'theano.tensor.var.TensorVariable'>
>>> type(y.owner.inputs[1].owner)
<class 'theano.gof.graph.Apply'>
>>> y.owner.inputs[1].owner.op
<theano.tensor.elemwise.DimShuffle object at 0x106fcaf10>
>>> y.owner.inputs[1].owner.inputs
[TensorConstant{2.0}]
```

Starting from this graph structure it is easier to understand how *automatic differentiation* proceeds and how the symbolic relations can be *optimized* for performance or stability.

Automatic Differentiation

Having the graph structure, computing automatic differentiation is simple. The only thing `tensor.grad()` has to do is to traverse the graph from the outputs back towards the inputs through all *apply* nodes (*apply* nodes are those that define which computations the graph does). For each such *apply* node, its *op* defines how to compute the *gradient* of the node's outputs with respect to its inputs. Note that if an *op* does not provide this information, it is assumed that the *gradient* is not defined. Using the chain rule these gradients can be composed in order to obtain the expression of the *gradient* of the graph's output with respect to the graph's inputs .

A following section of this tutorial will examine the topic of *differentiation* in greater detail.

Optimizations

When compiling a Theano function, what you give to the `theano.function` is actually a graph (starting from the output variables you can traverse the graph up to the input variables). While this graph structure shows how to compute the output from the input, it also offers the possibility to improve the way this computation is carried out. The way optimizations work in Theano is by identifying and replacing certain patterns in the graph with other specialized patterns that produce the same results but are either faster or more stable. Optimizations can also detect identical subgraphs and ensure that the same values are not computed twice or reformulate parts of the graph to a GPU specific version.

For example, one (simple) optimization that Theano uses is to replace the pattern $\frac{xy}{y}$ by x.

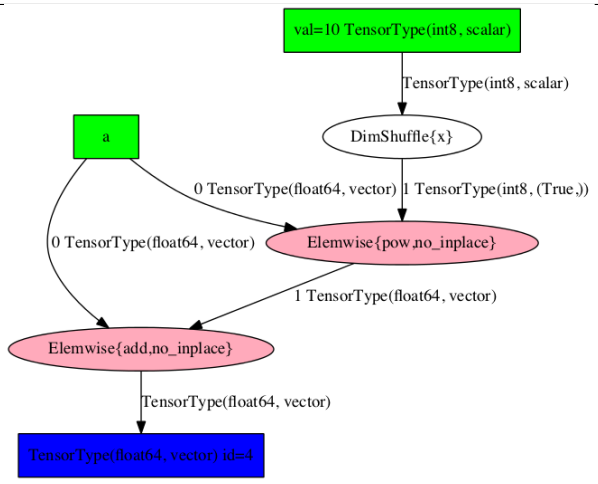
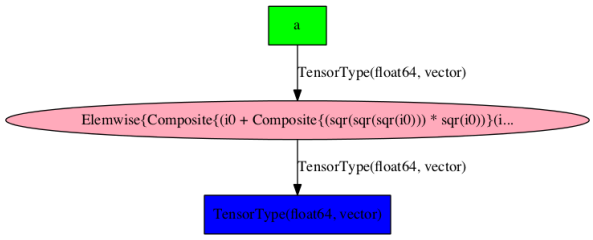
Further information regarding the optimization *process* and the specific *optimizations* that are applicable is respectively available in the library and on the entrance page of the documentation.

Example

Symbolic programming involves a change of paradigm: it will become clearer as we apply it. Consider the following example of optimization:

```
>>> import theano
>>> a = theano.tensor.vector("a")      # declare symbolic variable
>>> b = a + a ** 10                    # build symbolic expression
>>> f = theano.function([a], b)        # compile function
>>> print f([0, 1, 2])                 # prints `array([0,2,1026])`
[    0.     2.  1026.]
>>> theano.printing.pydotprint(b, outfile="./pics/symbolic_graph_unopt.png", var_with_name_
The output file is available at ./pics/symbolic_graph_unopt.png
>>> theano.printing.pydotprint(f, outfile="./pics/symbolic_graph_opt.png", var_with_name_si
The output file is available at ./pics/symbolic_graph_opt.png
```

We used `theano.printing.pydotprint()` to visualize the optimized graph (right), which is much more compact than the unoptimized graph (left).

Unoptimized graph	Optimized graph
	

Printing/Drawing Theano graphs

Theano provides the functions `theano.printing.pprint()` and `theano.printing.debugprint()` to print a graph to the terminal before or after compilation. `pprint()` is more compact and math-like, `debugprint()` is more verbose. Theano also provides `pydotprint()` that creates an image of the function. You can read about them in *printing – Graph Printing and Symbolic Print Statement*.

Note: When printing Theano functions, they can sometimes be hard to read. To help with this, you can disable some Theano optimizations by using the Theano flag: `optimizer_excluding=fusion:inplace`. Do not use this during real job execution, as this will make the graph slower and use more memory.

Consider again the logistic regression example:

```
>>> import numpy
>>> import theano
>>> import theano.tensor as T
>>> rng = numpy.random
>>> # Training data
>>> N = 400
>>> feats = 784
>>> D = (rng.randn(N, feats).astype(theano.config.floatX), rng.randint(size=N,low=0, high=2
>>> training_steps = 10000
>>> # Declare Theano symbolic variables
>>> x = T.matrix("x")
>>> y = T.vector("y")
>>> w = theano.shared(rng.randn(feats).astype(theano.config.floatX), name="w")
>>> b = theano.shared(numpy.asarray(0., dtype=theano.config.floatX), name="b")
>>> x.tag.test_value = D[0]
>>> y.tag.test_value = D[1]
>>> # Construct Theano expression graph
>>> p_1 = 1 / (1 + T.exp(-T.dot(x, w)-b)) # Probability of having a one
>>> prediction = p_1 > 0.5 # The prediction that is done: 0 or 1
>>> # Compute gradients
>>> xent = -y*T.log(p_1) - (1-y)*T.log(1-p_1) # Cross-entropy
>>> cost = xent.mean() + 0.01*(w**2).sum() # The cost to optimize
>>> gw,gb = T.grad(cost, [w,b])
>>> # Training and prediction function
>>> train = theano.function(inputs=[x,y], outputs=[prediction, xent], updates=[[w, w-0.01*g
>>> predict = theano.function(inputs=[x], outputs=prediction, name = "predict")
```

Pretty Printing

```
>>> theano.printing.pprint(prediction)
'gt((TensorConstant{1} / (TensorConstant{1} + exp(((-(x \\dot w)) - b)))),
TensorConstant{0.5})'
```

Debug Print

The pre-compilation graph:

```
>>> theano.printing.debugprint(prediction)
    Elemwise{gt,no_inplace} [@A] ''
    |Elemwise{true_div,no_inplace} [@B] ''
    | |DimShuffle{x} [@C] ''
    | | |TensorConstant{1} [@D]
    | |Elemwise{add,no_inplace} [@E] ''
    |   |DimShuffle{x} [@F] ''
    |   | |TensorConstant{1} [@D]
    |   |Elemwise{exp,no_inplace} [@G] ''
    |     |Elemwise{sub,no_inplace} [@H] ''
    |       |Elemwise{neg,no_inplace} [@I] ''
    |       | |dot [@J] ''
    |       |   |x [@K]
    |       |   |w [@L]
    |       |DimShuffle{x} [@M] ''
    |         |b [@N]
    |DimShuffle{x} [@O] ''
      |TensorConstant{0.5} [@P]
```

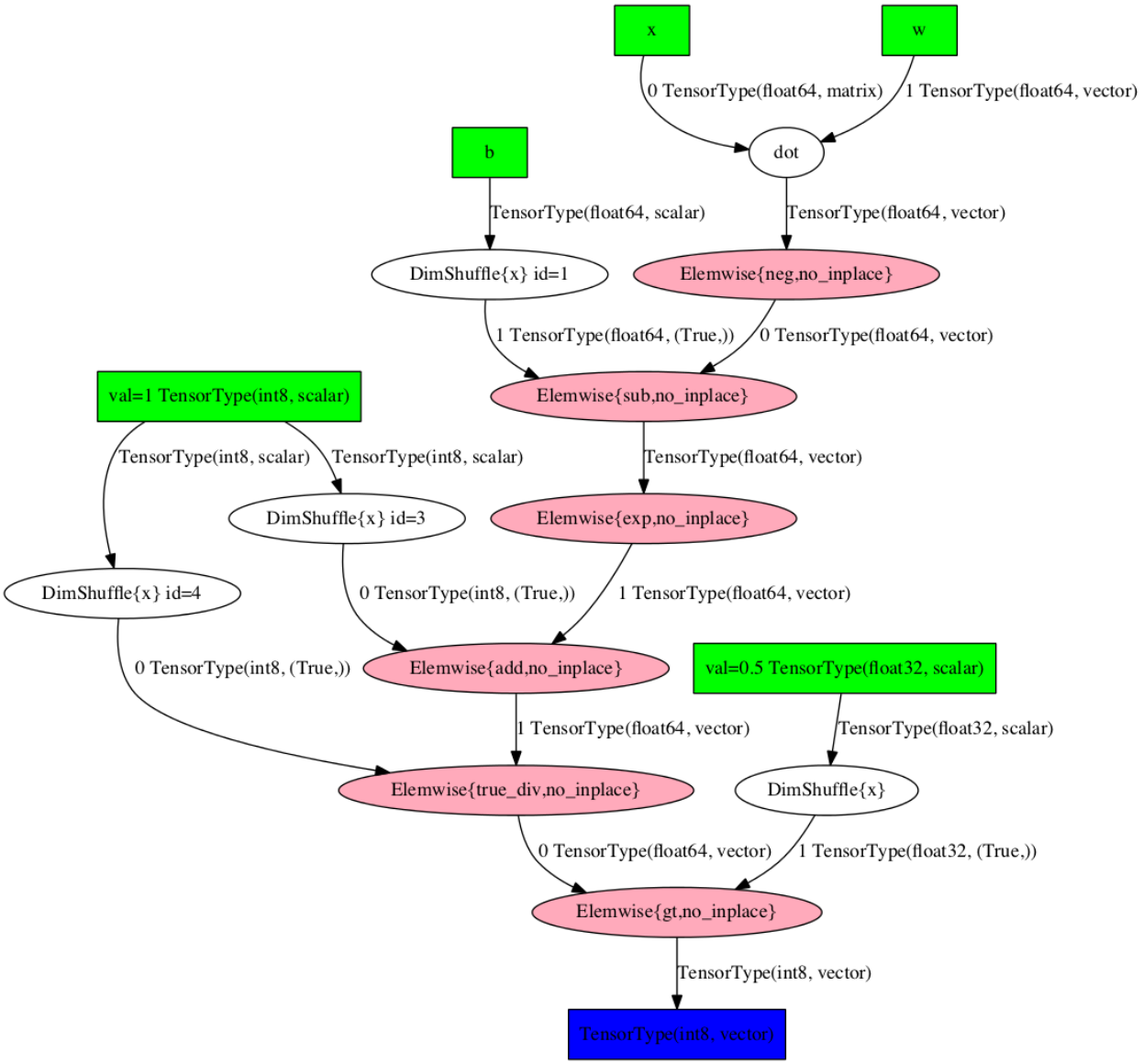
The post-compilation graph:

```
>>> theano.printing.debugprint(predict)
    Elemwise{Composite{GT(scalar_sigmoid((-((-i0) - i1))), i2)}} [@A] ''   4
     |CGemv{inplace} [@B] ''   3
     | |Alloc [@C] ''   2
     | | |TensorConstant{0.0} [@D]
     | | |Shape_i{0} [@E] ''   1
     | |   |x [@F]
     | |TensorConstant{1.0} [@G]
     | |x [@F]
     | |w [@H]
     | |TensorConstant{0.0} [@D]
     |InplaceDimShuffle{x} [@I] ''   0
     | |b [@J]
     |TensorConstant{(1,) of 0.5} [@K]
```

Picture Printing of Graphs

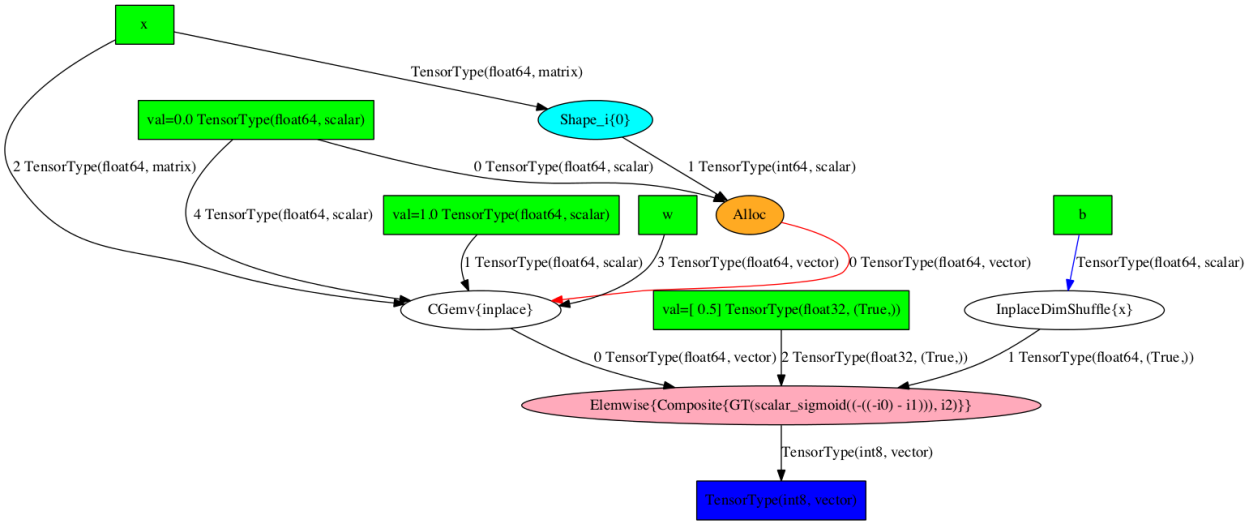
The pre-compilation graph:

```
>>> theano.printing.pydotprint(prediction, outfile="pics/logreg_pydotprint_prediction.png",
The output file is available at pics/logreg_pydotprint_prediction.png
```



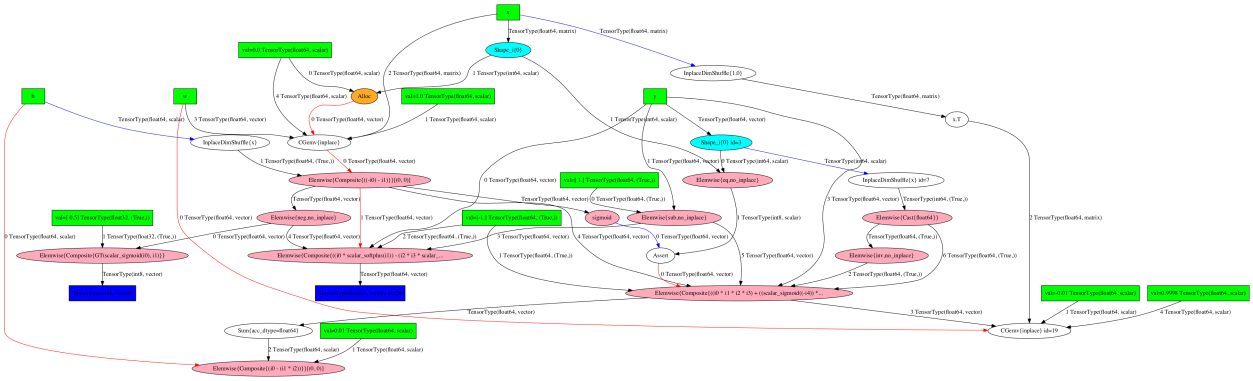
The post-compilation graph:

```
>>> theano.printing.pydotprint(predict, outfile="pics/logreg_pydotprint_predict.png", var_w
The output file is available at pics/logreg_pydotprint_predict.png
```



The optimized training graph:

```
>>> theano.printing.pydotprint(train, outfile="pics/logreg_pydotprint_train.png", var_with_
The output file is available at pics/logreg_pydotprint_train.png
```



Derivatives in Theano

Computing Gradients

Now let's use Theano for a slightly more sophisticated task: create a function which computes the derivative of some expression y with respect to its parameter x. To do this we will use the macro `T.grad`. For instance, we can compute the gradient of x^2 with respect to x. Note that: $d(x^2)/dx = 2 \cdot x$.

Here is the code to compute this gradient:

```
>>> from theano import pp
>>> x = T.dscalar('x')
>>> y = x ** 2
>>> gy = T.grad(y, x)
>>> pp(gy)  # print out the gradient prior to optimization
'((fill((x ** 2), 1.0) * 2) * (x ** (2 - 1)))'
>>> f = function([x], gy)
>>> f(4)
```

```
array(8.0)
>>> f(94.2)
array(188.40000000000001)
```

In this example, we can see from `pp(gy)` that we are computing the correct symbolic gradient. `fill((x ** 2), 1.0)` means to make a matrix of the same shape as *x ** 2* and fill it with *1.0*.

Note: The optimizer simplifies the symbolic gradient expression. You can see this by digging inside the internal properties of the compiled function.

```
pp(f.maker.fgraph.outputs[0])
'(2.0 * x)'
```

After optimization there is only one Apply node left in the graph, which doubles the input.

We can also compute the gradient of complex expressions such as the logistic function defined above. It turns out that the derivative of the logistic is: $ds(x)/dx = s(x) \cdot (1 - s(x))$.

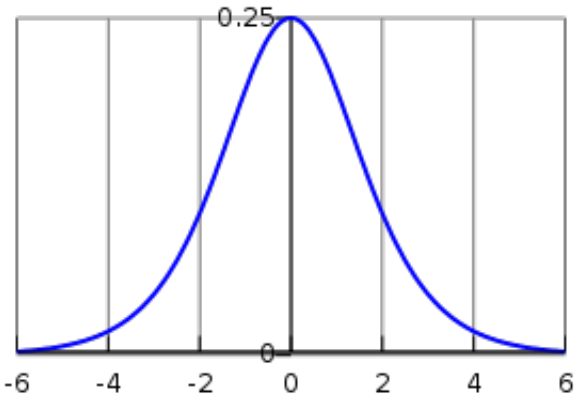


Figure 6.3: A plot of the gradient of the logistic function, with *x* on the x-axis and $ds(x)/dx$ on the y-axis.

```
>>> x = T.dmatrix('x')
>>> s = T.sum(1 / (1 + T.exp(-x)))
>>> gs = T.grad(s, x)
>>> dlogistic = function([x], gs)
>>> dlogistic([[0, 1], [-1, -2]])
array([[ 0.25      ,  0.19661193],
       [ 0.19661193,  0.10499359]])
```

In general, for any **scalar** expression *s*, `T.grad(s, w)` provides the Theano expression for computing $\frac{\partial s}{\partial w}$. In this way Theano can be used for doing **efficient** symbolic differentiation (as the expression returned by `T.grad` will be optimized during compilation), even for function with many inputs. (see automatic differentiation for a description of symbolic differentiation).

Note: The second argument of `T.grad` can be a list, in which case the output is also a list. The order in both lists is important: element *i* of the output list is the gradient of the first argument of `T.grad` with respect to the *i*-th element of the list given as second argument. The first argument of `T.grad` has to be a

scalar (a tensor of size 1). For more information on the semantics of the arguments of `T.grad` and details about the implementation, see *this* section of the library.

Additional information on the inner workings of differentiation may also be found in the more advanced tutorial *Extending Theano*.

Computing the Jacobian

In Theano's parlance, the term *Jacobian* designates the tensor comprising the first partial derivatives of the output of a function with respect to its inputs. (This is a generalization of to the so-called Jacobian matrix in Mathematics.) Theano implements the `theano.gradient.jacobian()` macro that does all that is needed to compute the Jacobian. The following text explains how to do it manually.

In order to manually compute the Jacobian of some function *y* with respect to some parameter *x* we need to use `scan`. What we do is to loop over the entries in *y* and compute the gradient of *y[i]* with respect to *x*.

Note: `scan` is a generic op in Theano that allows writing in a symbolic manner all kinds of recurrent equations. While creating symbolic loops (and optimizing them for performance) is a hard task, effort is being done for improving the performance of `scan`. We shall return to *scan* later in this tutorial.

```
>>> x = T.dvector('x')
>>> y = x ** 2
>>> J, updates = theano.scan(lambda i, y,x : T.grad(y[i], x), sequences=T.arange(y.shape[0
>>> f = function([x], J, updates=updates)
>>> f([4, 4])
array([[ 8.,  0.],
       [ 0.,  8.]])
```

What we do in this code is to generate a sequence of *ints* from *0* to `y.shape[0]` using `T.arange`. Then we loop through this sequence, and at each step, we compute the gradient of element *y[i]* with respect to *x*. `scan` automatically concatenates all these rows, generating a matrix which corresponds to the Jacobian.

Note: There are some pitfalls to be aware of regarding `T.grad`. One of them is that you cannot rewrite the above expression of the Jacobian as `theano.scan(lambda y_i,x: T.grad(y_i,x), sequences=y, non_sequences=x)`, even though from the documentation of scan this seems possible. The reason is that *y_i* will not be a function of *x* anymore, while *y[i]* still is.

Computing the Hessian

In Theano, the term *Hessian* has the usual mathematical acception: It is the matrix comprising the second order partial derivative of a function with scalar output and vector input. Theano implements `theano.gradient.hessian()` macro that does all that is needed to compute the Hessian. The following text explains how to do it manually.

You can compute the Hessian manually similarly to the Jacobian. The only difference is that now, instead of computing the Jacobian of some expression *y*, we compute the Jacobian of `T.grad(cost,x)`, where *cost* is some scalar.

```
>>> x = T.dvector('x')
>>> y = x ** 2
>>> cost = y.sum()
>>> gy = T.grad(cost, x)
>>> H, updates = theano.scan(lambda i, gy,x : T.grad(gy[i], x), sequences=T.arange(gy.shape
>>> f = function([x], H, updates=updates)
>>> f([4, 4])
array([[ 2.,  0.],
       [ 0.,  2.]])
```

Jacobian times a Vector

Sometimes we can express the algorithm in terms of Jacobians times vectors, or vectors times Jacobians. Compared to evaluating the Jacobian and then doing the product, there are methods that compute the desired results while avoiding actual evaluation of the Jacobian. This can bring about significant performance gains. A description of one such algorithm can be found here:

- Barak A. Pearlmutter, "Fast Exact Multiplication by the Hessian", *Neural Computation, 1994*

While in principle we would want Theano to identify these patterns automatically for us, in practice, implementing such optimizations in a generic manner is extremely difficult. Therefore, we provide special functions dedicated to these tasks.

R-operator The *R operator* is built to evaluate the product between a Jacobian and a vector, namely $\frac{\partial f(x)}{\partial x} v$. The formulation can be extended even for x being a matrix, or a tensor in general, case in which also the Jacobian becomes a tensor and the product becomes some kind of tensor product. Because in practice we end up needing to compute such expressions in terms of weight matrices, Theano supports this more generic form of the operation. In order to evaluate the *R-operation* of expression y, with respect to x, multiplying the Jacobian with v you need to do something similar to this:

```
>>> W = T.dmatrix('W')
>>> V = T.dmatrix('V')
>>> x = T.dvector('x')
>>> y = T.dot(x, W)
>>> JV = T.Rop(y, W, V)
>>> f = theano.function([W, V, x], JV)
>>> f([[1, 1], [1, 1]], [[2, 2], [2, 2]], [0,1])
array([ 2.,  2.])
```

List of Op that implement Rop.

L-operator In similitude to the *R-operator*, the *L-operator* would compute a *row* vector times the Jacobian. The mathematical formula would be $v \frac{\partial f(x)}{\partial x}$. The *L-operator* is also supported for generic tensors (not only for vectors). Similarly, it can be implemented as follows:

```
>>> W = T.dmatrix('W')
>>> v = T.dvector('v')
>>> x = T.dvector('x')
>>> y = T.dot(x, W)
```

```
>>> VJ = T.Lop(y, W, v)
>>> f = theano.function([v,x], VJ)
>>> f([2, 2], [0, 1])
array([[ 0.,  0.],
       [ 2.,  2.]])
```

Note: *v*, the *point of evaluation*, differs between the *L-operator* and the *R-operator*. For the *L-operator*, the point of evaluation needs to have the same shape as the output, whereas for the *R-operator* this point should have the same shape as the input parameter. Furthermore, the results of these two operations differ. The result of the *L-operator* is of the same shape as the input parameter, while the result of the *R-operator* has a shape similar to that of the output.

Hessian times a Vector

If you need to compute the *Hessian times a vector*, you can make use of the above-defined operators to do it more efficiently than actually computing the exact Hessian and then performing the product. Due to the symmetry of the Hessian matrix, you have two options that will give you the same result, though these options might exhibit differing performances. Hence, we suggest profiling the methods before using either one of the two:

```
>>> x = T.dvector('x')
>>> v = T.dvector('v')
>>> y = T.sum(x ** 2)
>>> gy = T.grad(y, x)
>>> vH = T.grad(T.sum(gy * v), x)
>>> f = theano.function([x, v], vH)
>>> f([4, 4], [2, 2])
array([ 4.,  4.])
```

or, making use of the *R-operator*:

```
>>> x = T.dvector('x')
>>> v = T.dvector('v')
>>> y = T.sum(x ** 2)
>>> gy = T.grad(y, x)
>>> Hv = T.Rop(gy, x, v)
>>> f = theano.function([x, v], Hv)
>>> f([4, 4], [2, 2])
array([ 4.,  4.])
```

Final Pointers

- The `grad` function works symbolically: it receives and returns Theano variables.
- `grad` can be compared to a macro since it can be applied repeatedly.
- Scalar costs only can be directly handled by `grad`. Arrays are handled through repeated applications.
- Built-in functions allow to compute efficiently *vector times Jacobian* and *vector times Hessian*.

- Work is in progress on the optimizations required to compute efficiently the full Jacobian and the Hessian matrix as well as the *Jacobian times vector*.

Configuration Settings and Compiling Modes

Configuration

The `config` module contains several *attributes* that modify Theano's behavior. Many of these attributes are examined during the import of the `theano` module and several are assumed to be read-only.

As a rule, the attributes in the `config` *module should not be modified inside the user code.*

Theano's code comes with default values for these attributes, but you can override them from your `.theanorc` file, and override those values in turn by the `THEANO_FLAGS` environment variable.

The order of precedence is:

1. an assignment to theano.config.<property>
2. an assignment in `THEANO_FLAGS`
3. an assignment in the .theanorc file (or the file indicated in `THEANORC`)

You can display the current/effective configuration at any time by printing theano.config. For example, to see a list of all active configuration variables, type this from the command-line:

```
python -c 'import theano; print theano.config' | less
```

For more detail, see *Configuration* in the library.

Exercise

Consider the logistic regression:

```
import numpy
import theano
import theano.tensor as T
rng = numpy.random

N = 400
feats = 784
D = (rng.randn(N, feats).astype(theano.config.floatX),
rng.randint(size=N,low=0, high=2).astype(theano.config.floatX))
training_steps = 10000

# Declare Theano symbolic variables
x = T.matrix("x")
y = T.vector("y")
w = theano.shared(rng.randn(feats).astype(theano.config.floatX), name="w")
b = theano.shared(numpy.asarray(0., dtype=theano.config.floatX), name="b")
x.tag.test_value = D[0]
y.tag.test_value = D[1]
```

```
#print "Initial model:"
#print w.get_value(), b.get_value()

# Construct Theano expression graph
p_1 = 1 / (1 + T.exp(-T.dot(x, w)-b)) # Probability of having a one
prediction = p_1 > 0.5 # The prediction that is done: 0 or 1
xent = -y*T.log(p_1) - (1-y)*T.log(1-p_1) # Cross-entropy
cost = xent.mean() + 0.01*(w**2).sum() # The cost to optimize
gw,gb = T.grad(cost, [w,b])

# Compile expressions to functions
train = theano.function(
            inputs=[x,y],
            outputs=[prediction, xent],
            updates={w:w-0.01*gw, b:b-0.01*gb},
            name = "train")
predict = theano.function(inputs=[x], outputs=prediction,
            name = "predict")

if any([x.op.__class__.__name__ in ['Gemv', 'CGemv', 'Gemm', 'CGemm'] for x in
        train.maker.fgraph.toposort()]):
    print 'Used the cpu'
elif any([x.op.__class__.__name__ in ['GpuGemm', 'GpuGemv'] for x in
          train.maker.fgraph.toposort()]):
    print 'Used the gpu'
else:
    print 'ERROR, not able to tell if theano used the cpu or the gpu'
    print train.maker.fgraph.toposort()

for i in range(training_steps):
    pred, err = train(D[0], D[1])
#print "Final model:"
#print w.get_value(), b.get_value()

print "target values for D"
print D[1]

print "prediction on D"
print predict(D[0])
```

Modify and execute this example to run on CPU (the default) with floatX=float32 and time the execution using the command line `time python file.py`. Save your code as it will be useful later on.

Note:

- Apply the Theano flag `floatX=float32` (through `theano.config.floatX`) in your code.
- Cast inputs before storing them into a shared variable.
- Circumvent the automatic cast of *int32* with *float32* to *float64*:
  - Insert manual cast in your code or use *[u]int{8,16}*.
  - Insert manual cast around the mean operator (this involves division by length, which is an *int64*).

- Notice that a new casting mechanism is being developed.

Solution

Mode

Every time `theano.function` is called, the symbolic relationships between the input and output Theano *variables* are optimized and compiled. The way this compilation occurs is controlled by the value of the `mode` parameter.

Theano defines the following modes by name:

- `'FAST_COMPILE'`: Apply just a few graph optimizations and only use Python implementations.
- `'FAST_RUN'`: Apply all optimizations and use C implementations where possible.
- **`'DebugMode`: Verify the correctness of all optimizations, and compare C and Python** implementations. This mode can take much longer than the other modes, but can identify several kinds of problems.
- `'ProfileMode'` (deprecated): Same optimization as FAST_RUN, but print some profiling information.

The default mode is typically `FAST_RUN`, but it can be controlled via the configuration variable `config.mode`, which can be overridden by passing the keyword argument to `theano.function`.

short name	Full constructor	What does it do?
`FAST_COMPILE`	`theano.compile.mode.Mode(linker='py', optimizer='fast_compile')`	Python implementations only, quick and cheap graph transformations
`FAST_RUN`	`theano.compile.mode.Mode(linker='cvm', optimizer='fast_run')`	C implementations where available, all available graph transformations.
`DebugMode`	`theano.compile.debugmode.DebugMode()`	Both implementations where available, all available graph transformations.
`ProfileMode`	`theano.compile.profilemode.ProfileMode()`	Deprecated. C implementations where available, all available graph transformations, print profile information.

Note: For debugging purpose, there also exists a `MonitorMode` (which has no short name). It can be used to step through the execution of a function: see *the debugging FAQ* for details.

Linkers

A mode is composed of 2 things: an optimizer and a linker. Some modes, like `ProfileMode` and `DebugMode`, add logic around the optimizer and linker. `ProfileMode` and `DebugMode` use their own linker.

You can select which linker to use with the Theano flag `config.linker`. Here is a table to compare the different linkers.

linker	gc [1]	Raise error by op	Overhead	Definition
cvm	yes	yes	"++"	As c|py, but the runtime algo to execute the code is in c
cvm_nogc	no	yes	"+"	As cvm, but without gc
c|py [2]	yes	yes	"+++"	Try C code. If none exists for an op, use Python
c|py_nogc	no	yes	"++"	As c|py, but without gc
c	no	yes	"+"	Use only C code (if none available for an op, raise an error)
py	yes	yes	"+++"	Use only Python code
ProfileMode	no	no	"++++"	(Deprecated) Compute some extra profiling info
DebugMode	no	yes	VERY HIGH	Make many checks on what Theano computes

For more detail, see *Mode* in the library.

Using DebugMode

While normally you should use the `FAST_RUN` or `FAST_COMPILE` mode, it is useful at first (especially when you are defining new kinds of expressions or new optimizations) to run your code using the DebugMode (available via `mode='DebugMode`). The DebugMode is designed to run several self-checks and assertions that can help diagnose possible programming errors leading to incorrect output. Note that `DebugMode` is much slower than `FAST_RUN` or `FAST_COMPILE` so use it only during development (not when you launch 1000 processes on a cluster!).

DebugMode is used as follows:

```
x = T.dvector('x')

f = theano.function([x], 10 * x, mode='DebugMode')
```

[1] Garbage collection of intermediate results during computation. Otherwise, their memory space used by the ops is kept between Theano function calls, in order not to reallocate memory, and lower the overhead (make it faster...).

[2] Default

```
f([5])
f([0])
f([7])
```

If any problem is detected, DebugMode will raise an exception according to what went wrong, either at call time (*f(5)*) or compile time (`f = theano.function(x, 10 * x, mode='DebugMode')`). These exceptions should *not* be ignored; talk to your local Theano guru or email the users list if you cannot make the exception go away.

Some kinds of errors can only be detected for certain input value combinations. In the example above, there is no way to guarantee that a future call to, say *f(-1)*, won't cause a problem. DebugMode is not a silver bullet.

If you instantiate DebugMode using the constructor (see `DebugMode`) rather than the keyword `DebugMode` you can configure its behaviour via constructor arguments. The keyword version of DebugMode (which you get by using `mode='DebugMode'`) is quite strict.

For more detail, see *DebugMode* in the library.

ProfileMode

Note: ProfileMode is deprecated. Use `config.profile` instead.

Besides checking for errors, another important task is to profile your code. For this Theano uses a special mode called ProfileMode which has to be passed as an argument to `theano.function`. Using the ProfileMode is a three-step process.

Note: To switch the default accordingly, set the Theano flag `config.mode` to ProfileMode. In that case, when the Python process exits, it will automatically print the profiling information on the standard output.

The memory profile of the output of each `apply` node can be enabled with the Theano flag `config.ProfileMode.profile_memory`.

For more detail, see *ProfileMode* in the library.

Creating a ProfileMode Instance First create a ProfileMode instance:

```
>>> from theano import ProfileMode
>>> profmode = theano.ProfileMode(optimizer='fast_run', linker=theano.gof.OpWiseCLinker())
```

The ProfileMode constructor takes as input an optimizer and a linker. Which optimizer and linker to use will depend on the application. For example, a user wanting to profile the Python implementation only, should use the gof.PerformLinker (or "py" for short). On the other hand, a user wanting to profile his graph using C implementations wherever possible should use the `gof.OpWiseCLinker` (or "c|py"). For testing the speed of your code we would recommend using the `fast_run` optimizer and the `gof.OpWiseCLinker` linker.

Compiling your Graph with ProfileMode Once the ProfileMode instance is created, simply compile your graph as you would normally, by specifying the mode parameter.

```
>>> # with functions
>>> f = theano.function([input1,input2],[output1], mode=profmode)
```

Retrieving Timing Information Once your graph is compiled, simply run the program or operation you wish to profile, then call `profmode.print_summary()`. This will provide you with the desired timing information, indicating where your graph is spending most of its time. This is best shown through an example. Let's use our logistic regression example.

Compiling the module with `ProfileMode` and calling `profmode.print_summary()` generates the following output:

```
"""
ProfileMode.print_summary()
---------------------------

local_time 0.0749197006226 (Time spent running thunks)
Apply-wise summary: <fraction of local_time spent at this position> (<Apply position>, <App
        0.069   15      _dot22
        0.064   1       _dot22
        0.053   0       InplaceDimShuffle{x,0}
        0.049   2       InplaceDimShuffle{1,0}
        0.049   10      mul
        0.049   6       Elemwise{ScalarSigmoid{output_types_preference=<theano.scalar.basic
        0.049   3       InplaceDimShuffle{x}
        0.049   4       InplaceDimShuffle{x,x}
        0.048   14      Sum{0}
        0.047   7       sub
        0.046   17      mul
        0.045   9       sqr
        0.045   8       Elemwise{sub}
        0.045   16      Sum
        0.044   18      mul
   ... (remaining 6 Apply instances account for 0.25 of the runtime)
Op-wise summary: <fraction of local_time spent on this kind of Op> <Op name>
        0.139   * mul
        0.134   * _dot22
        0.092   * sub
        0.085   * Elemwise{Sub{output_types_preference=<theano.scalar.basic.transfer_type
        0.053   * InplaceDimShuffle{x,0}
        0.049   * InplaceDimShuffle{1,0}
        0.049   * Elemwise{ScalarSigmoid{output_types_preference=<theano.scalar.basic.trans
        0.049   * InplaceDimShuffle{x}
        0.049   * InplaceDimShuffle{x,x}
        0.048   * Sum{0}
        0.045   * sqr
        0.045   * Sum
        0.043   * Sum{1}
        0.042   * Elemwise{Mul{output_types_preference=<theano.scalar.basic.transfer_type
        0.041   * Elemwise{Add{output_types_preference=<theano.scalar.basic.transfer_type
        0.039   * Elemwise{Second{output_types_preference=<theano.scalar.basic.transfer_typ
```

```
   ... (remaining 0 Ops account for 0.00 of the runtime)
(*) Op is running a c implementation

"""
```

This output has two components. In the first section called *Apply-wise summary*, timing information is provided for the worst offending `Apply` nodes. This corresponds to individual op applications within your graph which took longest to execute (so if you use `dot` twice, you will see two entries there). In the second portion, the *Op-wise summary*, the execution time of all `Apply` nodes executing the same op are grouped together and the total execution time per op is shown (so if you use `dot` twice, you will see only one entry there corresponding to the sum of the time spent in each of them). Finally, notice that the `ProfileMode` also shows which ops were running a C implementation.

For more detail, see *ProfileMode* in the library.

Loading and Saving

Python's standard way of saving class instances and reloading them is the pickle mechanism. Many Theano objects can be *serialized* (and *deserialized*) by `pickle`, however, a limitation of `pickle` is that it does not save the code or data of a class along with the instance of the class being serialized. As a result, reloading objects created by a previous version of a class can be really problematic.

Thus, you will want to consider different mechanisms depending on the amount of time you anticipate between saving and reloading. For short-term (such as temp files and network transfers), pickling of the Theano objects or classes is possible. For longer-term (such as saving models from an experiment) you should not rely on pickled Theano objects; we recommend loading and saving the underlying shared objects as you would in the course of any other Python program.

The Basics of Pickling

The two modules `pickle` and `cPickle` have the same functionalities, but `cPickle`, coded in C, is much faster.

```
>>> import cPickle
```

You can serialize (or *save*, or *pickle*) objects to a file with `cPickle.dump`:

```
>>> f = file('obj.save', 'wb')
>>> cPickle.dump(my_obj, f, protocol=cPickle.HIGHEST_PROTOCOL)
>>> f.close()
```

Note: If you want your saved object to be stored efficiently, don't forget to use `cPickle.HIGHEST_PROTOCOL`. The resulting file can be dozens of times smaller than with the default protocol.

Note: Opening your file in binary mode (`'b'`) is required for portability (especially between Unix and Windows).

To de-serialize (or *load*, or *unpickle*) a pickled file, use `cPickle.load`:

```
>>> f = file('obj.save', 'rb')
>>> loaded_obj = cPickle.load(f)
>>> f.close()
```

You can pickle several objects into the same file, and load them all (in the same order):

```
>>> f = file('objects.save', 'wb')
>>> for obj in [obj1, obj2, obj3]:
>>>     cPickle.dump(obj, f, protocol=cPickle.HIGHEST_PROTOCOL)
>>> f.close()
```

Then:

```
>>> f = file('objects.save', 'rb')
>>> loaded_objects = []
>>> for i in range(3):
>>>     loaded_objects.append(cPickle.load(f))
>>> f.close()
```

For more details about pickle's usage, see Python documentation.

Short-Term Serialization

If you are confident that the class instance you are serializing will be deserialized by a compatible version of the code, pickling the whole model is an adequate solution. It would be the case, for instance, if you are saving models and reloading them during the same execution of your program, or if the class you're saving has been really stable for a while.

You can control what pickle will save from your object, by defining a __getstate__ method, and similarly __setstate__.

This will be especially useful if, for instance, your model class contains a link to the data set currently in use, that you probably don't want to pickle along every instance of your model.

For instance, you can define functions along the lines of:

```
def __getstate__(self):
    state = dict(self.__dict__)
    del state['training_set']
    return state

def __setstate__(self, d):
    self.__dict__.update(d)
    self.training_set = cPickle.load(file(self.training_set_file, 'rb'))
```

Robust Serialization

This type of serialization uses some helper functions particular to Theano. It serializes the object using Python's pickling protocol, but any `ndarray` or `CudaNdarray` objects contained within the object are saved separately as NPY files. These NPY files and the Pickled file are all saved together in single ZIP-file.

The main advantage of this approach is that you don't even need Theano installed in order to look at the values of shared variables that you pickled. You can just load the parameters manually with *numpy*.

```
numpy.load('model.zip')
```

This approach could be beneficial if you are sharing your model with people who might not have Theano installed, who are using a different Python version, or if you are planning to save your model for a long time (in which case version mismatches might make it difficult to unpickle objects).

See `theano.misc.pkl_utils.dump()` and `theano.misc.pkl_utils.load()`.

Long-Term Serialization

If the implementation of the class you want to save is quite unstable, for instance if functions are created or removed, class members are renamed, you should save and load only the immutable (and necessary) part of your class.

You can do that by defining __getstate__ and __setstate__ functions as above, maybe defining the attributes you want to save, rather than the ones you don't.

For instance, if the only parameters you want to save are a weight matrix W and a bias b, you can define:

```
def __getstate__(self):
    return (self.W, self.b)

def __setstate__(self, state):
    W, b = state
    self.W = W
    self.b = b
```

If at some point in time W is renamed to *weights* and b to *bias*, the older pickled files will still be usable, if you update these functions to reflect the change in name:

```
def __getstate__(self):
    return (self.weights, self.bias)

def __setstate__(self, state):
    W, b = state
    self.weights = W
    self.bias = b
```

For more information on advanced use of `pickle` and its internals, see Python's pickle documentation.

Conditions

IfElse vs Switch

- Both ops build a condition over symbolic variables.
- `IfElse` takes a *boolean* condition and two variables as inputs.

- `Switch` takes a *tensor* as condition and two variables as inputs. `switch` is an elementwise operation and is thus more general than `ifelse`.
- Whereas `switch` evaluates both *output* variables, `ifelse` is lazy and only evaluates one variable with respect to the condition.

Example

```
from theano import tensor as T
from theano.ifelse import ifelse
import theano, time, numpy

a,b = T.scalars('a', 'b')
x,y = T.matrices('x', 'y')

z_switch = T.switch(T.lt(a, b), T.mean(x), T.mean(y))
z_lazy = ifelse(T.lt(a, b), T.mean(x), T.mean(y))

f_switch = theano.function([a, b, x, y], z_switch,
                    mode=theano.Mode(linker='vm'))
f_lazyifelse = theano.function([a, b, x, y], z_lazy,
                    mode=theano.Mode(linker='vm'))

val1 = 0.
val2 = 1.
big_mat1 = numpy.ones((10000, 1000))
big_mat2 = numpy.ones((10000, 1000))

n_times = 10

tic = time.clock()
for i in xrange(n_times):
    f_switch(val1, val2, big_mat1, big_mat2)
print 'time spent evaluating both values %f sec' % (time.clock() - tic)

tic = time.clock()
for i in xrange(n_times):
    f_lazyifelse(val1, val2, big_mat1, big_mat2)
print 'time spent evaluating one value %f sec' % (time.clock() - tic)
```

In this example, the `IfElse` op spends less time (about half as much) than `Switch` since it computes only one variable out of the two.

```
>>> python ifelse_switch.py
time spent evaluating both values 0.6700 sec
time spent evaluating one value 0.3500 sec
```

Unless `linker='vm'` or `linker='cvm'` are used, `ifelse` will compute both variables and take the same computation time as `switch`. Although the linker is not currently set by default to `cvm`, it will be in the near future.

There is no automatic optimization replacing a `switch` with a broadcasted scalar to an `ifelse`, as this is not always faster. See this ticket.

Loop

Scan

- A general form of *recurrence*, which can be used for looping.
- *Reduction* and *map* (loop over the leading dimensions) are special cases of `scan`.
- You `scan` a function along some input sequence, producing an output at each time-step.
- The function can see the *previous K time-steps* of your function.
- `sum()` could be computed by scanning the *z + x(i)* function over a list, given an initial state of *z=0*.
- Often a *for* loop can be expressed as a `scan()` operation, and `scan` is the closest that Theano comes to looping.
- Advantages of using `scan` over *for* loops:
  - Number of iterations to be part of the symbolic graph.
  - Minimizes GPU transfers (if GPU is involved).
  - Computes gradients through sequential steps.
  - Slightly faster than using a *for* loop in Python with a compiled Theano function.
  - Can lower the overall memory usage by detecting the actual amount of memory needed.

The full documentation can be found in the library: *Scan*.

Scan Example: Computing tanh(x(t).dot(W) + b) elementwise

```
import theano
import theano.tensor as T
import numpy as np

# defining the tensor variables
X = T.matrix("X")
W = T.matrix("W")
b_sym = T.vector("b_sym")

results, updates = theano.scan(lambda v: T.tanh(T.dot(v, W) + b_sym), sequences=X)
compute_elementwise = theano.function(inputs=[X, W, b_sym], outputs=[results])

# test values
x = np.eye(2, dtype=theano.config.floatX)
w = np.ones((2, 2), dtype=theano.config.floatX)
b = np.ones((2), dtype=theano.config.floatX)
b[1] = 2

print compute_elementwise(x, w, b)[0]

# comparison with numpy
print np.tanh(x.dot(w) + b)
```

Scan Example: Computing the sequence x(t) = tanh(x(t - 1).dot(W) + y(t).dot(U) + p(T - t).dot(V))

```
import theano
import theano.tensor as T
import numpy as np

# define tensor variables
X = T.vector("X")
W = T.matrix("W")
b_sym = T.vector("b_sym")
U = T.matrix("U")
Y = T.matrix("Y")
V = T.matrix("V")
P = T.matrix("P")

results, updates = theano.scan(lambda y, p, x_tm1: T.tanh(T.dot(x_tm1, W) + T.dot(y, U) + T
          sequences=[Y, P[::-1]], outputs_info=[X])
compute_seq = theano.function(inputs=[X, W, Y, U, P, V], outputs=[results])

# test values
x = np.zeros((2), dtype=theano.config.floatX)
x[1] = 1
w = np.ones((2, 2), dtype=theano.config.floatX)
y = np.ones((5, 2), dtype=theano.config.floatX)
y[0, :] = -3
u = np.ones((2, 2), dtype=theano.config.floatX)
p = np.ones((5, 2), dtype=theano.config.floatX)
p[0, :] = 3
v = np.ones((2, 2), dtype=theano.config.floatX)

print compute_seq(x, w, y, u, p, v)[0]

# comparison with numpy
x_res = np.zeros((5, 2), dtype=theano.config.floatX)
x_res[0] = np.tanh(x.dot(w) + y[0].dot(u) + p[4].dot(v))
for i in range(1, 5):
  x_res[i] = np.tanh(x_res[i - 1].dot(w) + y[i].dot(u) + p[4-i].dot(v))
print x_res
```

Scan Example: Computing norms of lines of X

```
import theano
import theano.tensor as T
import numpy as np

# define tensor variable
X = T.matrix("X")
results, updates = theano.scan(lambda x_i: T.sqrt((x_i ** 2).sum()), sequences=[X])
compute_norm_lines = theano.function(inputs=[X], outputs=[results])

# test value
x = np.diag(np.arange(1, 6, dtype=theano.config.floatX), 1)
print compute_norm_lines(x)[0]

# comparison with numpy
```

```
print np.sqrt((x ** 2).sum(1))
```

Scan Example: Computing norms of columns of X

```
import theano
import theano.tensor as T
import numpy as np

# define tensor variable
X = T.matrix("X")
results, updates = theano.scan(lambda x_i: T.sqrt((x_i ** 2).sum()), sequences=[X.T])
compute_norm_cols = theano.function(inputs=[X], outputs=[results])

# test value
x = np.diag(np.arange(1, 6, dtype=theano.config.floatX), 1)
print compute_norm_cols(x)[0]

# comparison with numpy
print np.sqrt((x ** 2).sum(0))
```

Scan Example: Computing trace of X

```
import theano
import theano.tensor as T
import numpy as np
floatX = "float32"

# define tensor variable
X = T.matrix("X")
results, updates = theano.scan(lambda i, j, t_f: T.cast(X[i, j] + t_f, floatX),
                  sequences=[T.arange(X.shape[0]), T.arange(X.shape[1])],
                  outputs_info=np.asarray(0., dtype=floatX))
result = results[-1]
compute_trace = theano.function(inputs=[X], outputs=[result])

# test value
x = np.eye(5, dtype=theano.config.floatX)
x[0] = np.arange(5, dtype=theano.config.floatX)
print compute_trace(x)[0]

# comparison with numpy
print np.diagonal(x).sum()
```

Scan Example: Computing the sequence x(t) = x(t - 2).dot(U) + x(t - 1).dot(V) + tanh(x(t - 1).dot(W) + b)

```
import theano
import theano.tensor as T
import numpy as np

# define tensor variables
X = T.matrix("X")
W = T.matrix("W")
b_sym = T.vector("b_sym")
```

```
U = T.matrix("U")
V = T.matrix("V")
n_sym = T.iscalar("n_sym")

results, updates = theano.scan(lambda x_tm2, x_tm1: T.dot(x_tm2, U) + T.dot(x_tm1, V) + T.t
                    n_steps=n_sym, outputs_info=[dict(initial=X, taps=[-2, -1])])
compute_seq2 = theano.function(inputs=[X, U, V, W, b_sym, n_sym], outputs=[results])

# test values
x = np.zeros((2, 2), dtype=theano.config.floatX) # the initial value must be able to retur
x[1, 1] = 1
w = 0.5 * np.ones((2, 2), dtype=theano.config.floatX)
u = 0.5 * (np.ones((2, 2), dtype=theano.config.floatX) - np.eye(2, dtype=theanc.config.floa
v = 0.5 * np.ones((2, 2), dtype=theano.config.floatX)
n = 10
b = np.ones((2), dtype=theano.config.floatX)

print compute_seq2(x, u, v, w, b, n)

# comparison with numpy
x_res = np.zeros((10, 2))
x_res[0] = x[0].dot(u) + x[1].dot(v) + np.tanh(x[1].dot(w) + b)
x_res[1] = x[1].dot(u) + x_res[0].dot(v) + np.tanh(x_res[0].dot(w) + b)
x_res[2] = x_res[0].dot(u) + x_res[1].dot(v) + np.tanh(x_res[1].dot(w) + b)
for i in range(2, 10):
  x_res[i] = (x_res[i - 2].dot(u) + x_res[i - 1].dot(v) +
              np.tanh(x_res[i - 1].dot(w) + b))
print x_res
```

Scan Example: Computing the Jacobian of y = tanh(v.dot(A)) wrt x

```
import theano
import theano.tensor as T
import numpy as np

# define tensor variables
v = T.vector()
A = T.matrix()
y = T.tanh(T.dot(v, A))
results, updates = theano.scan(lambda i: T.grad(y[i], v), sequences=[T.arange(y.shape[0])])
compute_jac_t = theano.function([A, v], [results], allow_input_downcast=True) # shape (d_ou

# test values
x = np.eye(5, dtype=theano.config.floatX)[0]
w = np.eye(5, 3, dtype=theano.config.floatX)
w[2] = np.ones((3), dtype=theano.config.floatX)
print compute_jac_t(w, x)[0]

# compare with numpy
print ((1 - np.tanh(x.dot(w)) ** 2) * w).T
```

Note that we need to iterate over the indices of `y` and not over the elements of `y`. The reason is that scan create a placeholder variable for its internal function and this placeholder variable does not have the same

dependencies than the variables that will replace it.

Scan Example: Accumulate number of loop during a scan

```
import theano
import theano.tensor as T
import numpy as np

# define shared variables
k = theano.shared(0)
n_sym = T.iscalar("n_sym")

results, updates = theano.scan(lambda:{k:(k + 1)}, n_steps=n_sym)
accumulator = theano.function([n_sym], [], updates=updates, allow_input_downcast=True)

k.get_value()
accumulator(5)
k.get_value()
```

Scan Example: Computing tanh(v.dot(W) + b) * d where d is binomial

```
import theano
import theano.tensor as T
import numpy as np

# define tensor variables
X = T.matrix("X")
W = T.matrix("W")
b_sym = T.vector("b_sym")

# define shared random stream
trng = T.shared_randomstreams.RandomStreams(1234)
d=trng.binomial(size=W[1].shape)

results, updates = theano.scan(lambda v: T.tanh(T.dot(v, W) + b_sym) * d, sequences=X)
compute_with_bnoise = theano.function(inputs=[X, W, b_sym], outputs=[results],
                          updates=updates, allow_input_downcast=True)
x = np.eye(10, 2, dtype=theano.config.floatX)
w = np.ones((2, 2), dtype=theano.config.floatX)
b = np.ones((2), dtype=theano.config.floatX)

print compute_with_bnoise(x, w, b)
```

Note that if you want to use a random variable `d` that will not be updated through scan loops, you should pass this variable as a `non_sequences` arguments.

Scan Example: Computing pow(A, k)

```
import theano
import theano.tensor as T
theano.config.warn.subtensor_merge_bug = False

k = T.iscalar("k")
A = T.vector("A")
```

```
def inner_fct(prior_result, B):
    return prior_result * B

# Symbolic description of the result
result, updates = theano.scan(fn=inner_fct,
                              outputs_info=T.ones_like(A),
                              non_sequences=A, n_steps=k)

# Scan has provided us with A ** 1 through A ** k.  Keep only the last
# value. Scan notices this and does not waste memory saving them.
final_result = result[-1]

power = theano.function(inputs=[A, k], outputs=final_result,
                        updates=updates)

print power(range(10), 2)
#[  0.   1.   4.   9.  16.  25.  36.  49.  64.  81.]
```

Scan Example: Calculating a Polynomial

```
import numpy
import theano
import theano.tensor as T
theano.config.warn.subtensor_merge_bug = False

coefficients = theano.tensor.vector("coefficients")
x = T.scalar("x")
max_coefficients_supported = 10000

# Generate the components of the polynomial
full_range=theano.tensor.arange(max_coefficients_supported)
components, updates = theano.scan(fn=lambda coeff, power, free_var:
                                     coeff * (free_var ** power),
                                  outputs_info=None,
                                  sequences=[coefficients, full_range],
                                  non_sequences=x)

polynomial = components.sum()
calculate_polynomial = theano.function(inputs=[coefficients, x],
                                       outputs=polynomial)

test_coeff = numpy.asarray([1, 0, 2], dtype=numpy.float32)
print calculate_polynomial(test_coeff, 3)
# 19.0
```

Exercise

Run both examples.

Modify and execute the polynomial example to have the reduction done by `scan`.

`Solution`

Sparse

In general, *sparse* matrices provide the same functionality as regular matrices. The difference lies in the way the elements of *sparse* matrices are represented and stored in memory. Only the non-zero elements of the latter are stored. This has some potential advantages: first, this may obviously lead to reduced memory usage and, second, clever storage methods may lead to reduced computation time through the use of sparse specific algorithms. We usually refer to the generically stored matrices as *dense* matrices.

Theano's sparse package provides efficient algorithms, but its use is not recommended in all cases or for all matrices. As an obvious example, consider the case where the *sparsity proportion* if very low. The *sparsity proportion* refers to the ratio of the number of zero elements to the number of all elements in a matrix. A low sparsity proportion may result in the use of more space in memory since not only the actual data is stored, but also the position of nearly every element of the matrix. This would also require more computation time whereas a dense matrix representation along with regular optimized algorithms might do a better job. Other examples may be found at the nexus of the specific purpose and structure of the matrices. More documentation may be found in the SciPy Sparse Reference.

Since sparse matrices are not stored in contiguous arrays, there are several ways to represent them in memory. This is usually designated by the so-called `format` of the matrix. Since Theano's sparse matrix package is based on the SciPy sparse package, complete information about sparse matrices can be found in the SciPy documentation. Like SciPy, Theano does not implement sparse formats for arrays with a number of dimensions different from two.

So far, Theano implements two `formats` of sparse matrix: `csc` and `csr`. Those are almost identical except that `csc` is based on the *columns* of the matrix and `csr` is based on its *rows*. They both have the same purpose: to provide for the use of efficient algorithms performing linear algebra operations. A disadvantage is that they fail to give an efficient way to modify the sparsity structure of the underlying matrix, i.e. adding new elements. This means that if you are planning to add new elements in a sparse matrix very often in your computational graph, perhaps a tensor variable could be a better choice.

More documentation may be found in the *Sparse Library Reference*.

Before going further, here are the `import` statements that are assumed for the rest of the tutorial:

```
>>> import theano
>>> import numpy as np
>>> import scipy.sparse as sp
>>> from theano import sparse
```

Compressed Sparse Format

Theano supports two *compressed sparse formats* `csc` and `csr`, respectively based on columns and rows. They have both the same attributes: `data`, `indices`, `indptr` and `shape`.

- The `data` attribute is a one-dimentionnal `ndarray` which contains all the non-zero elements of the sparse matrix.
- The `indices` and `indptr` attributes are used to store the position of the data in the sparse matrix.
- The `shape` attribute is exactly the same as the `shape` attribute of a dense (i.e. generic) matrix. It can be explicitly specified at the creation of a sparse matrix if it cannot be infered from the first three

attributes.

Which format should I use? At the end, the format does not affect the length of the `data` and `indices` attributes. They are both completly fixed by the number of elements you want to store. The only thing that changes with the format is `indptr`. In `csc` format, the matrix is compressed along columns so a lower number of columns will result in less memory use. On the other hand, with the `csr` format, the matrix is compressed along the rows and with a matrix that have a lower number of rows, `csr` format is a better choice. So here is the rule:

Note: If shape[0] > shape[1], use `csr` format. Otherwise, use `csc`.

Sometimes, since the sparse module is young, ops does not exist for both format. So here is what may be the most relevent rule:

Note: Use the format compatible with the ops in your computation graph.

The documentation about the ops and their supported format may be found in the *Sparse Library Reference*.

Handling Sparse in Theano

Most of the ops in Theano depend on the `format` of the sparse matrix. That is why there are two kinds of constructors of sparse variables: `csc_matrix` and `csr_matrix`. These can be called with the usual `name` and `dtype` parameters, but no `broadcastable` flags are allowed. This is forbidden since the sparse package, as the SciPy sparse module, does not provide any way to handle a number of dimensions different from two. The set of all accepted `dtype` for the sparse matrices can be found in `sparse.all_dtypes`.

```
>>> sparse.all_dtypes
set(['int8', 'int16', 'int32', 'int64', 'uint8', 'uint16', 'uint32', 'uint64',
     'float32', 'float64', 'complex64', 'complex128'])
```

To and Fro To move back and forth from a dense matrix to a sparse matrix representation, Theano provides the `dense_from_sparse`, `csr_from_dense` and `csc_from_dense` functions. No additional detail must be provided. Here is an example that performs a full cycle from sparse to sparse:

```
>>> x = sparse.csc_matrix(name='x', dtype='float32')
>>> y = sparse.dense_from_sparse(x)
>>> z = sparse.csc_from_dense(y)
```

Properties and Construction Although sparse variables do not allow direct access to their properties, this can be accomplished using the `csm_properties` function. This will return a tuple of one-dimensional `tensor` variables that represents the internal characteristics of the sparse matrix.

In order to reconstruct a sparse matrix from some properties, the functions `CSC` and `CSR` can be used. This will create the sparse matrix in the desired format. As an example, the following code reconstructs a `csc` matrix into a `csr` one.

```
>>> x = sparse.csc_matrix(name='x', dtype='int64')
>>> data, indices, indptr, shape = sparse.csm_properties(x)
>>> y = sparse.CSR(data, indices, indptr, shape)
>>> f = theano.function([x], y)
>>> a = sp.csc_matrix(np.asarray([[0, 1, 1], [0, 0, 0], [1, 0, 0]]))
>>> print a.toarray()
[[0 1 1]
 [0 0 0]
 [1 0 0]]
>>> print f(a).toarray()
[[0 0 1]
 [1 0 0]
 [1 0 0]]
```

The last example shows that one format can be obtained from transposition of the other. Indeed, when calling the `transpose` function, the sparse characteristics of the resulting matrix cannot be the same as the one provided as input.

Structured Operation Several ops are set to make use of the very peculiar structure of the sparse matrices. These ops are said to be *structured* and simply do not perform any computations on the zero elements of the sparse matrix. They can be thought as being applied only to the data attribute of the latter. Note that these structured ops provide a structured gradient. More explication below.

```
>>> x = sparse.csc_matrix(name='x', dtype='float32')
>>> y = sparse.structured_add(x, 2)
>>> f = theano.function([x], y)
>>> a = sp.csc_matrix(np.asarray([[0, 0, -1], [0, -2, 1], [3, 0, 0]], dtype='float32'))
>>> print a.toarray()
[[ 0.  0. -1.]
 [ 0. -2.  1.]
 [ 3.  0.  0.]]
>>> print f(a).toarray()
[[ 0.  0.  1.]
 [ 0.  0.  3.]
 [ 5.  0.  0.]]
```

Gradient The gradients of the ops in the sparse module can also be structured. Some ops provide a *flag* to indicate if the gradient is to be structured or not. The documentation can be used to determine if the gradient of an op is regular or structured or if its implementation can be modified. Similarly to structured ops, when a structured gradient is calculated, the computation is done only for the non-zero elements of the sparse matrix.

More documentation regarding the gradients of specific ops can be found in the *Sparse Library Reference*.

Using the GPU

For an introductory discussion of *Graphical Processing Units* (GPU) and their use for intensive parallel computation purposes, see GPGPU.

One of Theano's design goals is to specify computations at an abstract level, so that the internal function compiler has a lot of flexibility about how to carry out those computations. One of the ways we take advantage of this flexibility is in carrying out calculations on a graphics card.

There are two ways currently to use a gpu, one of which only supports NVIDIA cards (*CUDA backend*) and the other, in development, that should support any OpenCL device as well as NVIDIA cards (*GpuArray Backend*).

CUDA backend

If you have not done so already, you will need to install Nvidia's GPU-programming toolchain (CUDA) and configure Theano to use it. We provide installation instructions for *Linux*, *MacOS* and *Windows*.

Testing Theano with GPU To see if your GPU is being used, cut and paste the following program into a file and run it.

```
from theano import function, config, shared, sandbox
import theano.tensor as T
import numpy
import time

vlen = 10 * 30 * 768  # 10 x #cores x # threads per core
iters = 1000

rng = numpy.random.RandomState(22)
x = shared(numpy.asarray(rng.rand(vlen), config.floatX))
f = function([], T.exp(x))
print f.maker.fgraph.toposort()
t0 = time.time()
for i in xrange(iters):
    r = f()
t1 = time.time()
print 'Looping %d times took' % iters, t1 - t0, 'seconds'
print 'Result is', r
if numpy.any([isinstance(x.op, T.Elemwise) for x in f.maker.fgraph.toposort()]):
    print 'Used the cpu'
else:
    print 'Used the gpu'
```

The program just computes the `exp()` of a bunch of random numbers. Note that we use the `shared` function to make sure that the input *x* is stored on the graphics device.

If I run this program (in check1.py) with `device=cpu`, my computer takes a little over 3 seconds, whereas on the GPU it takes just over 0.64 seconds. The GPU will not always produce the exact same floating-point numbers as the CPU. As a benchmark, a loop that calls `numpy.exp(x.get_value())` takes about 46 seconds.

```
$ THEANO_FLAGS=mode=FAST_RUN,device=cpu,floatX=float32 python check1.py
[Elemwise{exp,no_inplace}(<TensorType(float32, vector)>)]
Looping 1000 times took 3.06635117531 seconds
Result is [ 1.23178029  1.61879337  1.52278066 ...,  2.20771813  2.29967761
```

```
  1.62323284]
Used the cpu

$ THEANO_FLAGS=mode=FAST_RUN,device=gpu,floatX=float32 python check1.py
Using gpu device 0: GeForce GTX 580
[GpuElemwise{exp,no_inplace}(<CudaNdarrayType(float32, vector)>), HostFromGpu(GpuElemwise{e
Looping 1000 times took 0.638810873032 seconds
Result is [ 1.23178029  1.61879349  1.52278066 ...,  2.20771813  2.29967761
  1.62323296]
Used the gpu
```

Note that GPU operations in Theano require for now `floatX` to be *float32* (see also below).

Returning a Handle to Device-Allocated Data The speedup is not greater in the preceding example because the function is returning its result as a NumPy ndarray which has already been copied from the device to the host for your convenience. This is what makes it so easy to swap in `device=gpu`, but if you don't mind less portability, you might gain a bigger speedup by changing the graph to express a computation with a GPU-stored result. The `gpu_from_host` op means "copy the input from the host to the GPU" and it is optimized away after the `T.exp(x)` is replaced by a GPU version of `exp()`.

```
from theano import function, config, shared, sandbox
import theano.sandbox.cuda.basic_ops
import theano.tensor as T
import numpy
import time

vlen = 10 * 30 * 768  # 10 x #cores x # threads per core
iters = 1000

rng = numpy.random.RandomState(22)
x = shared(numpy.asarray(rng.rand(vlen), config.floatX))
f = function([], sandbox.cuda.basic_ops.gpu_from_host(T.exp(x)))
print f.maker.fgraph.toposort()
t0 = time.time()
for i in xrange(iters):
    r = f()
t1 = time.time()
print 'Looping %d times took' % iters, t1 - t0, 'seconds'
print 'Result is', r
print 'Numpy result is', numpy.asarray(r)
if numpy.any([isinstance(x.op, T.Elemwise) for x in f.maker.fgraph.toposort()]):
    print 'Used the cpu'
else:
    print 'Used the gpu'
```

The output from this program is

```
$ THEANO_FLAGS=mode=FAST_RUN,device=gpu,floatX=float32 python check2.py
Using gpu device 0: GeForce GTX 580
[GpuElemwise{exp,no_inplace}(<CudaNdarrayType(float32, vector)>)]
Looping 1000 times took 0.34898686409 seconds
Result is <CudaNdarray object at 0x6a7a5f0>
```

```
Numpy result is [ 1.23178029  1.61879349  1.52278066 ...,  2.20771813  2.29967761
  1.62323296]
Used the gpu
```

Here we've shaved off about 50% of the run-time by simply not copying the resulting array back to the host. The object returned by each function call is now not a NumPy array but a "CudaNdarray" which can be converted to a NumPy ndarray by the normal NumPy casting mechanism using something like `numpy.asarray()`.

For even more speed you can play with the `borrow` flag. See *Borrowing when Constructing Function Objects*.

What Can Be Accelerated on the GPU The performance characteristics will change as we continue to optimize our implementations, and vary from device to device, but to give a rough idea of what to expect right now:

- Only computations with *float32* data-type can be accelerated. Better support for *float64* is expected in upcoming hardware but *float64* computations are still relatively slow (Jan 2010).
- Matrix multiplication, convolution, and large element-wise operations can be accelerated a lot (5-50x) when arguments are large enough to keep 30 processors busy.
- Indexing, dimension-shuffling and constant-time reshaping will be equally fast on GPU as on CPU.
- Summation over rows/columns of tensors can be a little slower on the GPU than on the CPU.
- Copying of large quantities of data to and from a device is relatively slow, and often cancels most of the advantage of one or two accelerated functions on that data. Getting GPU performance largely hinges on making data transfer to the device pay off.

Tips for Improving Performance on GPU

- Consider adding `floatX=float32` to your `.theanorc` file if you plan to do a lot of GPU work.
- Use the Theano flag `allow_gc=False`. See *GPU Async capabilities*
- Prefer constructors like `matrix`, `vector` and `scalar` to `dmatrix`, `dvector` and `dscalar` because the former will give you *float32* variables when `floatX=float32`.
- Ensure that your output variables have a *float32* dtype and not *float64*. The more *float32* variables are in your graph, the more work the GPU can do for you.
- Minimize tranfers to the GPU device by using `shared` *float32* variables to store frequently-accessed data (see `shared()`). When using the GPU, *float32* tensor `shared` variables are stored on the GPU by default to eliminate transfer time for GPU ops using those variables.
- If you aren't happy with the performance you see, try building your functions with `mode='ProfileMode'`. This should print some timing information at program termination. Is time being used sensibly? If an op or Apply is taking more time than its share, then if you know something about GPU programming, have a look at how it's implemented in theano.sandbox.cuda. Check the line similar to *Spent Xs(X%) in cpu op, Xs(X%) in gpu op and Xs(X%) in transfer op*. This can tell you if not enough of your graph is on the GPU or if there is too much memory transfer.

- Use nvcc options. nvcc supports those options to speed up some computations: *-ftz=true* to flush denormals values to zeros., *–prec-div=false* and *–prec-sqrt=false* options to speed up division and square root operation by being less precise. You can enable all of them with the *nvcc.flags=–use_fast_math* Theano flag or you can enable them individually as in this example: *nvcc.flags=-ftz=true –prec-div=false*.
- To investigate whether if all the Ops in the computational graph are running on GPU. It is possible to debug or check your code by providing a value to *assert_no_cpu_op* flag, i.e. *warn*, for warning *raise* for raising an error or *pdb* for putting a breakpoint in the computational graph if there is a CPU Op.

GPU Async capabilities Ever since Theano 0.6 we started to use the asynchronous capability of GPUs. This allows us to be faster but with the possibility that some errors may be raised later than when they should occur. This can cause difficulties when profiling Theano apply nodes. There is a NVIDIA driver feature to help with these issues. If you set the environment variable CUDA_LAUNCH_BLOCKING=1 then all kernel calls will be automatically synchronized. This reduces performance but provides good profiling and appropriately placed error messages.

This feature interacts with Theano garbage collection of intermediate results. To get the most of this feature, you need to disable the gc as it inserts synchronization points in the graph. Set the Theano flag `allow_gc=False` to get even faster speed! This will raise the memory usage.

Changing the Value of Shared Variables To change the value of a `shared` variable, e.g. to provide new data to processes, use `shared_variable.set_value(new_value)`. For a lot more detail about this, see *Understanding Memory Aliasing for Speed and Correctness*.

Exercise Consider again the logistic regression:

```
import numpy
import theano
import theano.tensor as T
rng = numpy.random

N = 400
feats = 784
D = (rng.randn(N, feats).astype(theano.config.floatX),
rng.randint(size=N,low=0, high=2).astype(theano.config.floatX))
training_steps = 10000

# Declare Theano symbolic variables
x = T.matrix("x")
y = T.vector("y")
w = theano.shared(rng.randn(feats).astype(theano.config.floatX), name="w")
b = theano.shared(numpy.asarray(0., dtype=theano.config.floatX), name="b")
x.tag.test_value = D[0]
y.tag.test_value = D[1]
#print "Initial model:"
#print w.get_value(), b.get_value()

# Construct Theano expression graph
p_1 = 1 / (1 + T.exp(-T.dot(x, w)-b)) # Probability of having a one
```

```
prediction = p_1 > 0.5 # The prediction that is done: 0 or 1
xent = -y*T.log(p_1) - (1-y)*T.log(1-p_1) # Cross-entropy
cost = xent.mean() + 0.01*(w**2).sum() # The cost to optimize
gw,gb = T.grad(cost, [w,b])

# Compile expressions to functions
train = theano.function(
            inputs=[x,y],
            outputs=[prediction, xent],
            updates={w:w-0.01*gw, b:b-0.01*gb},
            name = "train")
predict = theano.function(inputs=[x], outputs=prediction,
            name = "predict")

if any([x.op.__class__.__name__ in ['Gemv', 'CGemv', 'Gemm', 'CGemm'] for x in
        train.maker.fgraph.toposort()]):
    print 'Used the cpu'
elif any([x.op.__class__.__name__ in ['GpuGemm', 'GpuGemv'] for x in
          train.maker.fgraph.toposort()]):
    print 'Used the gpu'
else:
    print 'ERROR, not able to tell if theano used the cpu or the gpu'
    print train.maker.fgraph.toposort()

for i in range(training_steps):
    pred, err = train(D[0], D[1])
#print "Final model:"
#print w.get_value(), b.get_value()

print "target values for D"
print D[1]

print "prediction on D"
print predict(D[0])
```

Modify and execute this example to run on GPU with `floatX=float32` and time it using the command line `time python file.py`. (Of course, you may use some of your answer to the exercise in section *Configuration Settings and Compiling Mode*.)

Is there an increase in speed from CPU to GPU?

Where does it come from? (Use `ProfileMode`)

What can be done to further increase the speed of the GPU version? Put your ideas to test.

Note:

- Only 32 bit floats are currently supported (development is in progress).
- `Shared` variables with *float32* dtype are by default moved to the GPU memory space.
- There is a limit of one GPU per process.
- Use the Theano flag `device=gpu` to require use of the GPU device.

- Use `device=gpu{0, 1, ...}` to specify which GPU if you have more than one.
- Apply the Theano flag `floatX=float32` (through `theano.config.floatX`) in your code.
- `Cast` inputs before storing them into a `shared` variable.
- Circumvent the automatic cast of *int32* with *float32* to *float64*:
  - Insert manual cast in your code or use *[u]int{8,16}*.
  - Insert manual cast around the mean operator (this involves division by length, which is an *int64*).
  - Notice that a new casting mechanism is being developed.

`Solution`

GpuArray Backend

If you have not done so already, you will need to install libgpuarray as well as at least one computing toolkit. Instructions for doing so are provided at libgpuarray.

While all types of devices are supported if using OpenCL, for the remainder of this section, whatever compute device you are using will be referred to as GPU.

> **Warning:** While it is fully our intention to support OpenCL, as of May 2014 this support is still in its infancy. A lot of very useful ops still do not support it because they were ported from the old backend with minimal change.

Testing Theano with GPU To see if your GPU is being used, cut and paste the following program into a file and run it.

```
from theano import function, config, shared, tensor, sandbox
import numpy
import time

vlen = 10 * 30 * 768  # 10 x #cores x # threads per core
iters = 1000

rng = numpy.random.RandomState(22)
x = shared(numpy.asarray(rng.rand(vlen), config.floatX))
f = function([], tensor.exp(x))
print f.maker.fgraph.toposort()
t0 = time.time()
for i in xrange(iters):
    r = f()
t1 = time.time()
print 'Looping %d times took' % iters, t1 - t0, 'seconds'
print 'Result is', r
if numpy.any([isinstance(x.op, tensor.Elemwise) and
              ('Gpu' not in type(x.op).__name__)
```

```
            for x in f.maker.fgraph.toposort()]):
    print 'Used the cpu'
else:
    print 'Used the gpu'
```

The program just compute `exp()` of a bunch of random numbers. Note that we use the `theano.shared()` function to make sure that the input *x* is stored on the GPU.

```
$ THEANO_FLAGS=device=cpu python check1.py
[Elemwise{exp,no_inplace}(<TensorType(float64, vector)>)]
Looping 1000 times took 2.6071999073 seconds
Result is [ 1.23178032  1.61879341  1.52278065 ...,  2.20771815  2.29967753
  1.62323285]
Used the cpu

$ THEANO_FLAGS=device=cuda0 python check1.py
Using device cuda0: GeForce GTX 275
[GpuElemwise{exp,no_inplace}(<GpuArray<float64>>), HostFromGpu(gpuarray)(GpuElemwise{exp,no
Looping 1000 times took 2.28562092781 seconds
Result is [ 1.23178032  1.61879341  1.52278065 ...,  2.20771815  2.29967753
  1.62323285]
Used the gpu
```

Returning a Handle to Device-Allocated Data By default functions that execute on the GPU still return a standard numpy ndarray. A transfer operation is inserted just before the results are returned to ensure a consistent interface with CPU code. This allows changing the deivce some code runs on by only replacing the value of the `device` flag without touching the code.

If you don't mind a loss of flexibility, you can ask theano to return the GPU object directly. The following code is modifed to do just that.

```
from theano import function, config, shared, tensor, sandbox
import numpy
import time

vlen = 10 * 30 * 768  # 10 x #cores x # threads per core
iters = 1000

rng = numpy.random.RandomState(22)
x = shared(numpy.asarray(rng.rand(vlen), config.floatX))
f = function([], sandbox.gpuarray.basic_ops.gpu_from_host(tensor.exp(x)))
print f.maker.fgraph.toposort()
t0 = time.time()
for i in xrange(iters):
    r = f()
t1 = time.time()
print 'Looping %d times took' % iters, t1 - t0, 'seconds'
print 'Result is', numpy.asarray(r)
if numpy.any([isinstance(x.op, tensor.Elemwise) and
              ('Gpu' not in type(x.op).__name__)
              for x in f.maker.fgraph.toposort()]):
    print 'Used the cpu'
```

```
else:
    print 'Used the gpu'
```

Here the `theano.sandbox.gpuarray.basic.gpu_from_host()` call means "copy input to the GPU". However during the optimization phase, since the result will already be on th gpu, it will be removed. It is used here to tell theano that we want the result on the GPU.

The output is

```
$ THEANO_FLAGS=device=cuda0 python check2.py
Using device cuda0: GeForce GTX 275
[GpuElemwise{exp,no_inplace}(<GpuArray<float64>>)]
Looping 1000 times took 0.455810785294 seconds
Result is [ 1.23178032  1.61879341  1.52278065 ...,  2.20771815  2.29967753
  1.62323285]
Used the gpu
```

While the time per call appears to be much lower than the two previous invocations (and should indeed be lower, since we avoid a transfer) the massive speedup we obtained is in part due to asynchronous nature of execution on GPUs, meaning that the work isn't completed yet, just 'launched'. We'll talk about that later.

The object returned is a GpuArray from pygpu. It mostly acts as a numpy ndarray with some exceptions due to its data being on the GPU. You can copy it to the host and convert it to a regular ndarray by using usual numpy casting such as `numpy.asarray()`.

For even more speed, you can play with the `borrow` flag. See *Borrowing when Constructing Function Objects*.

What Can be Accelerated on the GPU The performance characteristics will of course vary from device to device, and also as we refine our implementation.

This backend supports all regular theano data types (float32, float64, int, ...) however GPU support varies and some units can't deal with double (float64) or small (less than 32 bits like int16) data types. You will get an error at compile time or runtime if this is the case.

Complex support is untested and most likely completely broken.

In general, large operations like matrix multiplication, or element-wise operations with large inputs, will be significatly faster.

GPU Async Capabilities By default, all operations on the GPU are run asynchronously. This means that they are only scheduled to run and the function returns. This is made somewhat transparently by the underlying libgpuarray.

A forced synchronization point is introduced when doing memory transfers between device and host. Another is introduced when releasing active memory buffers on the GPU (active buffers are buffers that are still in use by a kernel).

It is possible to force synchronization for a particular GpuArray by calling its `sync()` method. This is useful to get accurate timings when doing benchmarks.

The forced synchronization points interact with the garbage collection of the intermediate results. To get the fastest speed possible, you should disable the garbage collector by using the theano flag `allow_gc=False`. Be aware that this will increase memory usage sometimes significantly.

Software for Directly Programming a GPU

Leaving aside Theano which is a meta-programmer, there are:

- **CUDA**: GPU programming API by NVIDIA based on extension to C (CUDA C)
  - Vendor-specific
  - Numeric libraries (BLAS, RNG, FFT) are maturing.
- **OpenCL**: multi-vendor version of CUDA
  - More general, standardized.
  - Fewer libraries, lesser spread.
- **PyCUDA**: Python bindings to CUDA driver interface allow to access Nvidia's CUDA parallel computation API from Python
  - Convenience:

    Makes it easy to do GPU meta-programming from within Python.

    Abstractions to compile low-level CUDA code from Python (`pycuda.driver.SourceModule`).

    GPU memory buffer (`pycuda.gpuarray.GPUArray`).

    Helpful documentation.
  - Completeness: Binding to all of CUDA's driver API.
  - Automatic error checking: All CUDA errors are automatically translated into Python exceptions.
  - Speed: PyCUDA's base layer is written in C++.
  - Good memory management of GPU objects:

    Object cleanup tied to lifetime of objects (RAII, 'Resource Acquisition Is Initialization').

    Makes it much easier to write correct, leak- and crash-free code.

    PyCUDA knows about dependencies (e.g. it won't detach from a context before all memory allocated in it is also freed).

  (This is adapted from PyCUDA's documentation and Andreas Kloeckner's website on PyCUDA.)
- **PyOpenCL**: PyCUDA for OpenCL

Learning to Program with PyCUDA

If you already enjoy a good proficiency with the C programming language, you may easily leverage your knowledge by learning, first, to program a GPU with the CUDA extension to C (CUDA C) and, second, to use PyCUDA to access the CUDA API with a Python wrapper.

The following resources will assist you in this learning process:

- **CUDA API and CUDA C: Introductory**
  - NVIDIA's slides
  - Stein's (NYU) slides
- **CUDA API and CUDA C: Advanced**
  - MIT IAP2009 CUDA (full coverage: lectures, leading Kirk-Hwu textbook, examples, additional resources)
  - Course U. of Illinois (full lectures, Kirk-Hwu textbook)
  - NVIDIA's knowledge base (extensive coverage, levels from introductory to advanced)
  - practical issues (on the relationship between grids, blocks and threads; see also linked and related issues on same page)
  - CUDA optimisation
- **PyCUDA: Introductory**
  - Kloeckner's slides
  - Kloeckner' website
- **PYCUDA: Advanced**
  - PyCUDA documentation website

The following examples give a foretaste of programming a GPU with PyCUDA. Once you feel competent enough, you may try yourself on the corresponding exercises.

Example: PyCUDA

```
# (from PyCUDA's documentation)
import pycuda.autoinit
import pycuda.driver as drv
import numpy

from pycuda.compiler import SourceModule
mod = SourceModule("""
__global__ void multiply_them(float *dest, float *a, float *b)
{
  const int i = threadIdx.x;
  dest[i] = a[i] * b[i];
}
""")

multiply_them = mod.get_function("multiply_them")
```

```
a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)

dest = numpy.zeros_like(a)
multiply_them(
        drv.Out(dest), drv.In(a), drv.In(b),
        block=(400,1,1), grid=(1,1))

assert numpy.allclose(dest, a*b)
print dest
```

Exercise Run the preceding example.

Modify and execute to work for a matrix of shape (20, 10). **Example: Theano + PyCUDA**

```
import numpy, theano
import theano.misc.pycuda_init
from pycuda.compiler import SourceModule
import theano.sandbox.cuda as cuda

class PyCUDADoubleOp(theano.Op):

    __props__ = ()

    def make_node(self, inp):
        inp = cuda.basic_ops.gpu_contiguous(
           cuda.basic_ops.as_cuda_ndarray_variable(inp))
        assert inp.dtype == "float32"
        return theano.Apply(self, [inp], [inp.type()])

    def make_thunk(self, node, storage_map, _, _2):
        mod = SourceModule("""
    __global__ void my_fct(float * i0, float * o0, int size) {
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if(i<size){
        o0[i] = i0[i]*2;
    }
  }""")
        pycuda_fct = mod.get_function("my_fct")
        inputs = [storage_map[v] for v in node.inputs]
        outputs = [storage_map[v] for v in node.outputs]

        def thunk():
            z = outputs[0]
            if z[0] is None or z[0].shape != inputs[0][0].shape:
                z[0] = cuda.CudaNdarray.zeros(inputs[0][0].shape)
            grid = (int(numpy.ceil(inputs[0][0].size / 512.)), 1)
            pycuda_fct(inputs[0][0], z[0], numpy.intc(inputs[0][0].size),
                       block=(512, 1, 1), grid=grid)
        return thunk
```

Use this code to test it:

```
>>> x = theano.tensor.fmatrix()
>>> f = theano.function([x], PyCUDADoubleOp()(x))
>>> xv = numpy.ones((4, 5), dtype="float32")
>>> assert numpy.allclose(f(xv), xv*2)
>>> print numpy.asarray(f(xv))
```

Exercise Run the preceding example.

Modify and execute to multiply two matrices: *x* * *y*.

Modify and execute to return two outputs: *x* + *y* and *x* - *y*.

(Notice that Theano's current *elemwise fusion* optimization is only applicable to computations involving a single output. Hence, to gain efficiency over the basic solution that is asked here, the two operations would have to be jointly optimized explicitly in the code.)

Modify and execute to support *stride* (i.e. to avoid constraining the input to be *C-contiguous*).

Note

See *Other Implementations* to know how to handle random numbers on the GPU.

PyCUDA/CUDAMat/Gnumpy compatibility

PyCUDA

Currently, PyCUDA and Theano have different objects to store GPU data. The two implementations do not support the same set of features. Theano's implementation is called *CudaNdarray* and supports *strides*. It also only supports the *float32* dtype. PyCUDA's implementation is called *GPUArray* and doesn't support *strides*. However, it can deal with all NumPy and CUDA dtypes.

We are currently working on having the same base object for both that will also mimic Numpy. Until this is ready, here is some information on how to use both objects in the same script.

Transfer You can use the `theano.misc.pycuda_utils` module to convert GPUArray to and from CudaNdarray. The functions `to_cudandarray(x, copyif=False)` and `to_gpuarray(x)` return a new object that occupies the same memory space as the original. Otherwise it raises a *ValueError*. Because GPUArrays don't support strides, if the CudaNdarray is strided, we could copy it to have a non-strided copy. The resulting GPUArray won't share the same memory region. If you want this behavior, set `copyif=True` in `to_gpuarray`.

Compiling with PyCUDA You can use PyCUDA to compile CUDA functions that work directly on CudaNdarrays. Here is an example from the file `theano/misc/tests/test_pycuda_theano_simple.py`:

```
import sys
import numpy
import theano
import theano.sandbox.cuda as cuda_ndarray
import theano.misc.pycuda_init
import pycuda
import pycuda.driver as drv
import pycuda.gpuarray


def test_pycuda_theano():
    """Simple example with pycuda function and Theano CudaNdarray object."""
    from pycuda.compiler import SourceModule
    mod = SourceModule("""
__global__ void multiply_them(float *dest, float *a, float *b)
{
  const int i = threadIdx.x;
  dest[i] = a[i] * b[i];
}
""")

    multiply_them = mod.get_function("multiply_them")

    a = numpy.random.randn(100).astype(numpy.float32)
    b = numpy.random.randn(100).astype(numpy.float32)

    # Test with Theano object
    ga = cuda_ndarray.CudaNdarray(a)
    gb = cuda_ndarray.CudaNdarray(b)
    dest = cuda_ndarray.CudaNdarray.zeros(a.shape)
    multiply_them(dest, ga, gb,
                  block=(400, 1, 1), grid=(1, 1))
    assert (numpy.asarray(dest) == a * b).all()
```

Theano Op using a PyCUDA function You can use a GPU function compiled with PyCUDA in a Theano op:

```
import numpy, theano
import theano.misc.pycuda_init
from pycuda.compiler import SourceModule
import theano.sandbox.cuda as cuda

class PyCUDADoubleOp(theano.Op):
    __props__ = ()
    def make_node(self, inp):
        inp = cuda.basic_ops.gpu_contiguous(
           cuda.basic_ops.as_cuda_ndarray_variable(inp))
        assert inp.dtype == "float32"
        return theano.Apply(self, [inp], [inp.type()])
    def make_thunk(self, node, storage_map, _, _2):
        mod = SourceModule("""
    __global__ void my_fct(float * i0, float * o0, int size) {
```

```
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if(i<size){
        o0[i] = i0[i] * 2;
    }
  }""")
        pycuda_fct = mod.get_function("my_fct")
        inputs = [ storage_map[v] for v in node.inputs]
        outputs = [ storage_map[v] for v in node.outputs]
        def thunk():
            z = outputs[0]
            if z[0] is None or z[0].shape!=inputs[0][0].shape:
                z[0] = cuda.CudaNdarray.zeros(inputs[0][0].shape)
            grid = (int(numpy.ceil(inputs[0][0].size / 512.)),1)
            pycuda_fct(inputs[0][0], z[0], numpy.intc(inputs[0][0].size),
                       block=(512, 1, 1), grid=grid)
        thunk.lazy = False
        return thunk
```

CUDAMat

There are functions for conversion between CUDAMat objects and Theano's CudaNdArray objects. They obey the same principles as Theano's PyCUDA functions and can be found in `theano.misc.cudamat_utils.py`.

WARNING: There is a peculiar problem associated with stride/shape with those converters. In order to work, the test needs a *transpose* and *reshape*...

Gnumpy

There are conversion functions between Gnumpy *garray* objects and Theano CudaNdArray objects. They are also similar to Theano's PyCUDA functions and can be found in `theano.misc.gnumpy_utils.py`.

Understanding Memory Aliasing for Speed and Correctness

The aggressive reuse of memory is one of the ways through which Theano makes code fast, and it is important for the correctness and speed of your program that you understand how Theano might alias buffers.

This section describes the principles based on which Theano handles memory, and explains when you might want to alter the default behaviour of some functions and methods for faster performance.

The Memory Model: Two Spaces

There are some simple principles that guide Theano's handling of memory. The main idea is that there is a pool of memory managed by Theano, and Theano tracks changes to values in that pool.

- Theano manages its own memory space, which typically does not overlap with the memory of normal Python variables that non-Theano code creates.
- Theano functions only modify buffers that are in Theano's memory space.
- Theano's memory space includes the buffers allocated to store `shared` variables and the temporaries used to evaluate functions.
- Physically, Theano's memory space may be spread across the host, a GPU device(s), and in the future may even include objects on a remote machine.
- The memory allocated for a `shared` variable buffer is unique: it is never aliased to another `shared` variable.
- Theano's managed memory is constant while Theano functions are not running and Theano's library code is not running.
- The default behaviour of a function is to return user-space values for outputs, and to expect user-space values for inputs.

The distinction between Theano-managed memory and user-managed memory can be broken down by some Theano functions (e.g. `shared`, `get_value` and the constructors for `In` and `Out`) by using a `borrow=True` flag. This can make those methods faster (by avoiding copy operations) at the expense of risking subtle bugs in the overall program (by aliasing memory).

The rest of this section is aimed at helping you to understand when it is safe to use the `borrow=True` argument and reap the benefits of faster code.

Borrowing when Creating Shared Variables

A `borrow` argument can be provided to the shared-variable constructor.

```
import numpy, theano
np_array = numpy.ones(2, dtype='float32')

s_default = theano.shared(np_array)
s_false   = theano.shared(np_array, borrow=False)
s_true    = theano.shared(np_array, borrow=True)
```

By default (*s_default*) and when explicitly setting `borrow=False`, the shared variable we construct gets a [deep] copy of *np_array*. So changes we subsequently make to *np_array* have no effect on our shared variable.

```
np_array += 1 # now it is an array of 2.0 s

s_default.get_value()  # -> array([1.0, 1.0])
s_false.get_value()    # -> array([1.0, 1.0])
s_true.get_value()     # -> array([2.0, 2.0])
```

If we are running this with the CPU as the device, then changes we make to *np_array right away* will show up in `s_true.get_value()` because NumPy arrays are mutable, and *s_true* is using the *np_array* object as it's internal buffer.

However, this aliasing of *np_array* and *s_true* is not guaranteed to occur, and may occur only temporarily even if it occurs at all. It is not guaranteed to occur because if Theano is using a GPU device, then the `borrow` flag has no effect. It may occur only temporarily because if we call a Theano function that updates the value of *s_true* the aliasing relationship *may* or *may not* be broken (the function is allowed to update the `shared` variable by modifying its buffer, which will preserve the aliasing, or by changing which buffer the variable points to, which will terminate the aliasing).

Take home message:

It is a safe practice (and a good idea) to use `borrow=True` in a `shared` variable constructor when the `shared` variable stands for a large object (in terms of memory footprint) and you do not want to create copies of it in memory.

It is not a reliable technique to use `borrow=True` to modify `shared` variables through side-effect, because with some devices (e.g. GPU devices) this technique will not work.

Borrowing when Accessing Value of Shared Variables

Retrieving A `borrow` argument can also be used to control how a `shared` variable's value is retrieved.

```
s = theano.shared(np_array)

v_false = s.get_value(borrow=False) # N.B. borrow default is False
v_true = s.get_value(borrow=True)
```

When `borrow=False` is passed to `get_value`, it means that the return value may not be aliased to any part of Theano's internal memory. When `borrow=True` is passed to `get_value`, it means that the return value *might* be aliased to some of Theano's internal memory. But both of these calls might create copies of the internal memory.

The reason that `borrow=True` might still make a copy is that the internal representation of a `shared` variable might not be what you expect. When you create a `shared` variable by passing a NumPy array for example, then `get_value()` must return a NumPy array too. That's how Theano can make the GPU use transparent. But when you are using a GPU (or in the future perhaps a remote machine), then the numpy.ndarray is not the internal representation of your data. If you really want Theano to return its internal representation *and never copy it* then you should use the `return_internal_type=True` argument to `get_value`. It will never cast the internal object (always return in constant time), but might return various datatypes depending on contextual factors (e.g. the compute device, the dtype of the NumPy array).

```
v_internal = s.get_value(borrow=True, return_internal_type=True)
```

It is possible to use `borrow=False` in conjunction with `return_internal_type=True`, which will return a deep copy of the internal object. This is primarily for internal debugging, not for typical use.

For the transparent use of different type of optimization Theano can make, there is the policy that `get_value()` always return by default the same object type it received when the `shared` variable was created. So if you created manually data on the gpu and create a `shared` variable on the gpu with this data, `get_value` will always return gpu data even when `return_internal_type=False`.

Take home message:

It is safe (and sometimes much faster) to use `get_value(borrow=True)` when your code does not modify the return value. *Do not use this to modify a ''shared'' variable by side-effect* because it will make your code device-dependent. Modification of GPU variables through this sort of side-effect is impossible.

Assigning `Shared` variables also have a `set_value` method that can accept an optional `borrow=True` argument. The semantics are similar to those of creating a new `shared` variable - `borrow=False` is the default and `borrow=True` means that Theano *may* reuse the buffer you provide as the internal storage for the variable.

A standard pattern for manually updating the value of a `shared` variable is as follows:

```
s.set_value(
    some_inplace_fn(s.get_value(borrow=True)),
    borrow=True)
```

This pattern works regardless of the computing device, and when the latter makes it possible to expose Theano's internal variables without a copy, then it proceeds as fast as an in-place update.

When `shared` variables are allocated on the GPU, the transfers to and from the GPU device memory can be costly. Here are a few tips to ensure fast and efficient use of GPU memory and bandwidth:

- Prior to Theano 0.3.1, `set_value` did not work in-place on the GPU. This meant that, sometimes, GPU memory for the new value would be allocated before the old memory was released. If you're running near the limits of GPU memory, this could cause you to run out of GPU memory unnecessarily.

  Solution: update to a newer version of Theano.

- If you are going to swap several chunks of data in and out of a `shared` variable repeatedly, you will want to reuse the memory that you allocated the first time if possible - it is both faster and more memory efficient.

  Solution: upgrade to a recent version of Theano (>0.3.0) and consider padding your source data to make sure that every chunk is the same size.

- It is also worth mentioning that, current GPU copying routines support only contiguous memory. So Theano must make the value you provide *C-contiguous* prior to copying it. This can require an extra copy of the data on the host.

  Solution: make sure that the value you assign to a CudaNdarraySharedVariable is *already C-contiguous*.

(Further information on the current implementation of the GPU version of `set_value()` can be found here: *sandbox.cuda.var – The Variables for Cuda-allocated arrays*)

Borrowing when Constructing Function Objects

A `borrow` argument can also be provided to the `In` and `Out` objects that control how `theano.function` handles its argument[s] and return value[s].

```
import theano, theano.tensor

x = theano.tensor.matrix()
y = 2 * x
f = theano.function([theano.In(x, borrow=True)], theano.Out(y, borrow=True))
```

Borrowing an input means that Theano will treat the argument you provide as if it were part of Theano's pool of temporaries. Consequently, your input may be reused as a buffer (and overwritten!) during the computation of other variables in the course of evaluating that function (e.g. `f`).

Borrowing an output means that Theano will not insist on allocating a fresh output buffer every time you call the function. It will possibly reuse the same one as on a previous call, and overwrite the old content. Consequently, it may overwrite old return values through side-effect. Those return values may also be overwritten in the course of evaluating *another compiled function* (for example, the output may be aliased to a `shared` variable). So be careful to use a borrowed return value right away before calling any more Theano functions. The default is of course to *not borrow* internal results.

It is also possible to pass a `return_internal_type=True` flag to the `Out` variable which has the same interpretation as the `return_internal_type` flag to the `shared` variable's `get_value` function. Unlike `get_value()`, the combination of `return_internal_type=True` and `borrow=True` arguments to `Out()` are not guaranteed to avoid copying an output value. They are just hints that give more flexibility to the compilation and optimization of the graph.

For GPU graphs, this borrowing can have a major speed impact. See the following code:

```
from theano import function, config, shared, sandbox, tensor, Out
import numpy
import time

vlen = 10 * 30 * 768  # 10 x # cores x # threads per core
iters = 1000

rng = numpy.random.RandomState(22)
x = shared(numpy.asarray(rng.rand(vlen), config.floatX))
f1 = function([], sandbox.cuda.basic_ops.gpu_from_host(tensor.exp(x)))
f2 = function([],
              Out(sandbox.cuda.basic_ops.gpu_from_host(tensor.exp(x)),
                  borrow=True))
t0 = time.time()
for i in xrange(iters):
    r = f1()
t1 = time.time()
no_borrow = t1 - t0
t0 = time.time()
for i in xrange(iters):
    r = f2()
t1 = time.time()
print 'Looping', iters, 'times took', no_borrow, 'seconds without borrow',
print 'and', t1 - t0, 'seconds with borrow.'
if numpy.any([isinstance(x.op, tensor.Elemwise) and
              ('Gpu' not in type(x.op).__name__)
              for x in f1.maker.fgraph.toposort()]):
    print 'Used the cpu'
```

```
else:
    print 'Used the gpu'
```

Which produces this output:

```
$ THEANO_FLAGS=device=gpu0,floatX=float32 python test1.py
Using gpu device 0: GeForce GTX 275
Looping 1000 times took 0.368273973465 seconds without borrow and 0.0240728855133 seconds w
Used the gpu
```

Take home message:

When an input *x* to a function is not needed after the function returns and you would like to make it available to Theano as additional workspace, then consider marking it with `In(x, borrow=True)`. It may make the function faster and reduce its memory requirement. When a return value *y* is large (in terms of memory footprint), and you only need to read from it once, right away when it's returned, then consider marking it with an `Out(y, borrow=True)`.

How Shape Information is Handled by Theano

It is not possible to strictly enforce the shape of a Theano variable when building a graph since the particular value provided at run-time for a parameter of a Theano function may condition the shape of the Theano variables in its graph.

Currently, information regarding shape is used in two ways in Theano:

- To generate faster C code for the 2d convolution on the CPU and the GPU, when the exact output shape is known in advance.
- To remove computations in the graph when we only want to know the shape, but not the actual value of a variable. This is done with the Op.infer_shape method.

  Example:

```
>>> import theano
>>> x = theano.tensor.matrix('x')
>>> f = theano.function([x], (x ** 2).shape)
>>> theano.printing.debugprint(f)
MakeVector [@A] ''   2
 |Shape_i{0} [@B] ''   1
 | |x [@C]
 |Shape_i{1} [@D] ''   0
   |x [@C]
```

The output of this compiled function does not contain any multiplication or power. Theano has removed them to compute directly the shape of the output.

Shape Inference Problem

Theano propagates information about shape in the graph. Sometimes this can lead to errors. Consider this example:

```
>>> import numpy
>>> import theano
>>> x = theano.tensor.matrix('x')
>>> y = theano.tensor.matrix('y')
>>> z = theano.tensor.join(0, x, y)
>>> xv = numpy.random.rand(5, 4)
>>> yv = numpy.random.rand(3, 3)

>>> f = theano.function([x,y], z.shape)
>>> theano.printing.debugprint(f)
MakeVector [@A] ''   4
 |Elemwise{Add}[(0, 0)] [@B] ''   3
 | |Shape_i{0} [@C] ''   1
 | | |x [@D]
 | |Shape_i{0} [@E] ''   2
 |   |y [@F]
 |Shape_i{1} [@G] ''   0
   |x [@D]
```

print f(xv,yv)# DOES NOT RAISE AN ERROR AS SHOULD BE. [8, 4]

```
>>> f = theano.function([x,y], z)# Do not take the shape.
>>> theano.printing.debugprint(f)
Join [@A] ''   0
 |TensorConstant{0} [@B]
 |x [@C]
 |y [@D]

>>> f(xv,yv)
>>> # Raises a dimensions mismatch error.
```

As you can see, when asking only for the shape of some computation (`join` in the example), an inferred shape is computed directly, without executing the computation itself (there is no `join` in the first output or debugprint).

This makes the computation of the shape faster, but it can also hide errors. In this example, the computation of the shape of the output of `join` is done only based on the first input Theano variable, which leads to an error.

This might happen with other ops such as `elemwise` and `dot`, for example. Indeed, to perform some optimizations (for speed or stability, for instance), Theano assumes that the computation is correct and consistent in the first place, as it does here.

You can detect those problems by running the code without this optimization, using the Theano flag `optimizer_excluding=local_shape_to_shape_i`. You can also obtain the same effect by running in the modes `FAST_COMPILE` (it will not apply this optimization, nor most other optimizations) or `DebugMode` (it will test before and after all optimizations (much slower)).

Specifing Exact Shape

Currently, specifying a shape is not as easy and flexible as we wish and we plan some upgrade. Here is the current state of what can be done:

- You can pass the shape info directly to the `ConvOp` created when calling `conv2d`. You simply set the parameters `image_shape` and `filter_shape` inside the call. They must be tuples of 4 elements. For example:

```
theano.tensor.nnet.conv2d(..., image_shape=(7, 3, 5, 5), filter_shape=(2, 3, 4, 4))
```

- You can use the `SpecifyShape` op to add shape information anywhere in the graph. This allows to perform some optimizations. In the following example, this makes it possible to precompute the Theano function to a constant.

```
>>> import theano
>>> x = theano.tensor.matrix()
>>> x_specify_shape = theano.tensor.specify_shape(x, (2, 2))
>>> f = theano.function([x], (x_specify_shape ** 2).shape)
>>> theano.printing.debugprint(f)
DeepCopyOp [@A] ''   0
 |TensorConstant{(2,) of 2} [@B]
```

Future Plans

The parameter "constant shape" will be added to `theano.shared()`. This is probably the most frequent occurrence with `shared` variables. It will make the code simpler and will make it possible to check that the shape does not change when updating the `shared` variable.

Debugging Theano: FAQ and Troubleshooting

There are many kinds of bugs that might come up in a computer program. This page is structured as a FAQ. It provides recipes to tackle common problems, and introduces some of the tools that we use to find problems in our own Theano code, and even (it happens) in Theano's internals, in *Using DebugMode*.

Isolating the Problem/Testing Theano Compiler

You can run your Theano function in a *DebugMode*. This tests the Theano optimizations and helps to find where NaN, inf and other problems come from.

Interpreting Error Messages

Even in its default configuration, Theano tries to display useful error messages. Consider the following faulty code.

```
import numpy as np
import theano
import theano.tensor as T

x = T.vector()
y = T.vector()
z = x + x
```

```
z = z + y
f = theano.function([x, y], z)
f(np.ones((2,)), np.ones((3,)))
```

Running the code above we see:

```
Traceback (most recent call last):
  File "test0.py", line 10, in <module>
    f(np.ones((2,)), np.ones((3,)))
  File "/PATH_TO_THEANO/theano/compile/function_module.py", line 605, in __call__
    self.fn.thunks[self.fn.position_of_error])
  File "/PATH_TO_THEANO/theano/compile/function_module.py", line 595, in __call__
    outputs = self.fn()
ValueError: Input dimension mis-match. (input[0].shape[0] = 3, input[1].shape[0] = 2)
Apply node that caused the error: Elemwise{add,no_inplace}(<TensorType(float64, vector)>, <
Inputs types: [TensorType(float64, vector), TensorType(float64, vector), TensorType(float64
Inputs shapes: [(3,), (2,), (2,)]
Inputs strides: [(8,), (8,), (8,)]
Inputs scalar values: ['not scalar', 'not scalar', 'not scalar']

HINT: Re-running with most Theano optimization disabled could give you a back-traces when t
HINT: Use the Theano flag 'exception_verbosity=high' for a debugprint of this apply node.
```

Arguably the most useful information is approximately half-way through the error message, where the kind of error is displayed along with its cause (*ValueError: Input dimension mis-match. (input[0].shape[0] = 3, input[1].shape[0] = 2*). Below it, some other information is given, such as the apply node that caused the error, as well as the input types, shapes, strides and scalar values.

The two hints can also be helpful when debugging. Using the theano flag `optimizer=fast_compile` or `optimizer=None` can often tell you the faulty line, while `exception_verbosity=high` will display a debugprint of the apply node. Using these hints, the end of the error message becomes :

```
Backtrace when the node is created:
  File "test0.py", line 8, in <module>
    z = z + y

Debugprint of the apply node:
Elemwise{add,no_inplace} [@A] <TensorType(float64, vector)> ''
 |Elemwise{add,no_inplace} [@B] <TensorType(float64, vector)> ''
 | |<TensorType(float64, vector)> [@C] <TensorType(float64, vector)>
 | |<TensorType(float64, vector)> [@C] <TensorType(float64, vector)>
 |<TensorType(float64, vector)> [@D] <TensorType(float64, vector)>
```

We can here see that the error can be traced back to the line `z = z + y`. For this example, using `optimizer=fast_compile` worked. If it did not, you could set `optimizer=None` or use test values.

Using Test Values

As of v.0.4.0, Theano has a new mechanism by which graphs are executed on-the-fly, before a `theano.function` is ever compiled. Since optimizations haven't been applied at this stage, it is easier for the user to locate the source of some bug. This functionality is enabled through the config flag `theano.config.compute_test_value`. Its use is best shown through the following example. Here,

we use `exception_verbosity=high` and `optimizer=fast_compile`, which would not tell you the line at fault. `optimizer=None` would and it could therefore be used instead of test values.

```
import numpy
import theano
import theano.tensor as T

# compute_test_value is 'off' by default, meaning this feature is inactive
theano.config.compute_test_value = 'off' # Use 'warn' to activate this feature

# configure shared variables
W1val = numpy.random.rand(2, 10, 10).astype(theano.config.floatX)
W1 = theano.shared(W1val, 'W1')
W2val = numpy.random.rand(15, 20).astype(theano.config.floatX)
W2 = theano.shared(W2val, 'W2')

# input which will be of shape (5,10)
x  = T.matrix('x')
# provide Theano with a default test-value
#x.tag.test_value = numpy.random.rand(5, 10)

# transform the shared variable in some way. Theano does not
# know off hand that the matrix func_of_W1 has shape (20, 10)
func_of_W1 = W1.dimshuffle(2, 0, 1).flatten(2).T

# source of error: dot product of 5x10 with 20x10
h1 = T.dot(x, func_of_W1)

# do more stuff
h2 = T.dot(h1, W2.T)

# compile and call the actual function
f = theano.function([x], h2)
f(numpy.random.rand(5, 10))
```

Running the above code generates the following error message:

```
Traceback (most recent call last):
  File "test1.py", line 31, in <module>
    f(numpy.random.rand(5, 10))
  File "PATH_TO_THEANO/theano/compile/function_module.py", line 605, in __call__
    self.fn.thunks[self.fn.position_of_error])
  File "PATH_TO_THEANO/theano/compile/function_module.py", line 595, in __call__
    outputs = self.fn()
ValueError: Shape mismatch: x has 10 cols (and 5 rows) but y has 20 rows (and 10 cols)
Apply node that caused the error: Dot22(x, DimShuffle{1,0}.0)
Inputs types: [TensorType(float64, matrix), TensorType(float64, matrix)]
Inputs shapes: [(5, 10), (20, 10)]
Inputs strides: [(80, 8), (8, 160)]
Inputs scalar values: ['not scalar', 'not scalar']

Debugprint of the apply node:
Dot22 [@A] <TensorType(float64, matrix)> ''
 |x [@B] <TensorType(float64, matrix)>
```

```
 |DimShuffle{1,0} [@C] <TensorType(float64, matrix)> ''
   |Flatten{2} [@D] <TensorType(float64, matrix)> ''
     |DimShuffle{2,0,1} [@E] <TensorType(float64, 3D)> ''
       |W1 [@F] <TensorType(float64, 3D)>

HINT: Re-running with most Theano optimization disabled could give you a back-traces when t
```

If the above is not informative enough, by instrumenting the code ever so slightly, we can get Theano to reveal the exact source of the error.

```
# enable on-the-fly graph computations
theano.config.compute_test_value = 'warn'

...

# input which will be of shape (5, 10)
x  = T.matrix('x')
# provide Theano with a default test-value
x.tag.test_value = numpy.random.rand(5, 10)
```

In the above, we are tagging the symbolic matrix *x* with a special test value. This allows Theano to evaluate symbolic expressions on-the-fly (by calling the `perform` method of each op), as they are being defined. Sources of error can thus be identified with much more precision and much earlier in the compilation pipeline. For example, running the above code yields the following error message, which properly identifies *line 24* as the culprit.

```
Traceback (most recent call last):
  File "test2.py", line 24, in <module>
    h1 = T.dot(x, func_of_W1)
  File "PATH_TO_THEANO/theano/tensor/basic.py", line 4734, in dot
    return _dot(a, b)
  File "PATH_TO_THEANO/theano/gof/op.py", line 545, in __call__
    required = thunk()
  File "PATH_TO_THEANO/theano/gof/op.py", line 752, in rval
    r = p(n, [x[0] for x in i], o)
  File "PATH_TO_THEANO/theano/tensor/basic.py", line 4554, in perform
    z[0] = numpy.asarray(numpy.dot(x, y))
ValueError: matrices are not aligned
```

The `compute_test_value` mechanism works as follows:

- Theano `constants` and `shared` variables are used as is. No need to instrument them.
- A Theano *variable* (i.e. `dmatrix`, `vector`, etc.) should be given a special test value through the attribute `tag.test_value`.
- Theano automatically instruments intermediate results. As such, any quantity derived from *x* will be given a `tag.test_value` automatically.

`compute_test_value` can take the following values:

- `off`: Default behavior. This debugging mechanism is inactive.
- `raise`: Compute test values on the fly. Any variable for which a test value is required, but not provided by the user, is treated as an error. An exception is raised accordingly.

- warn: Idem, but a warning is issued instead of an *Exception*.
- ignore: Silently ignore the computation of intermediate test values, if a variable is missing a test value.

Note: This feature is currently incompatible with Scan and also with ops which do not implement a perform method.

"How do I Print an Intermediate Value in a Function?"

Theano provides a 'Print' op to do this.

```
x = theano.tensor.dvector('x')

x_printed = theano.printing.Print('this is a very important value')(x)

f = theano.function([x], x * 5)
f_with_print = theano.function([x], x_printed * 5)

#this runs the graph without any printing
assert numpy.all( f([1, 2, 3]) == [5, 10, 15])

#this runs the graph with the message, and value printed
assert numpy.all( f_with_print([1, 2, 3]) == [5, 10, 15])
```

Since Theano runs your program in a topological order, you won't have precise control over the order in which multiple Print() ops are evaluted. For a more precise inspection of what's being computed where, when, and how, see the discussion *"How do I Step through a Compiled Function?"*.

Warning: Using this Print Theano Op can prevent some Theano optimization from being applied. This can also happen with stability optimization. So if you use this Print and have nan, try to remove them to know if this is the cause or not.

"How do I Print a Graph?" (before or after compilation)

Theano provides two functions (theano.pp() and theano.printing.debugprint()) to print a graph to the terminal before or after compilation. These two functions print expression graphs in different ways: pp() is more compact and math-like, debugprint() is more verbose. Theano also provides theano.printing.pydotprint() that creates a png image of the function.

You can read about them in *printing – Graph Printing and Symbolic Print Statement*.

"The Function I Compiled is Too Slow, what's up?"

First, make sure you're running in FAST_RUN mode. Even though FAST_RUN is the default mode, insist by passing mode='FAST_RUN' to theano.function (or theano.make) or by setting config.mode to FAST_RUN.

Second, try the Theano *ProfileMode*. This will tell you which `Apply` nodes, and which ops are eating up your CPU cycles.

Tips:

- Use the flags `floatX=float32` to require type *float32* instead of *float64*; Use the Theano constructors matrix(),vector(),... instead of dmatrix(), dvector(),... since they respectively involve the default types *float32* and *float64*.
- Check in the `profile` mode that there is no `Dot` op in the post-compilation graph while you are multiplying two matrices of the same type. `Dot` should be optimized to `dot22` when the inputs are matrices and of the same type. This can still happen when using `floatX=float32` when one of the inputs of the graph is of type *float64*.

"Why does my GPU function seem to be slow?"

When you compile a theano function, if you do not get the speedup that you expect over the CPU performance of the same code. It is oftentimes due to the fact that some Ops might be running on CPU instead GPU. If that is the case, you can use assert_no_cpu_op to check if there is a CPU Op on your computational graph. assert_no_cpu_op can take the following one of the three options:

- `warn`: Raise a warning
- `pdb`: Stop with a pdb in the computational graph during the compilation
- `raise`: Raise an error, if there is a CPU Op in the computational graph.

It is possible to use this mode by providing the flag in THEANO_FLAGS, such as: `THEANO_FLAGS="float32,device=gpu,assert_no_cpu_op='raise'" python test.py`

But note that this optimization will not catch all the CPU Ops, it might miss some Ops.

"How do I Step through a Compiled Function?"

You can use `MonitorMode` to inspect the inputs and outputs of each node being executed when the function is called. The code snipped below shows how to print all inputs and outputs:

```
import theano

def inspect_inputs(i, node, fn):
    print i, node, "input(s) value(s):", [input[0] for input in fn.inputs],

def inspect_outputs(i, node, fn):
    print "output(s) value(s):", [output[0] for output in fn.outputs]

x = theano.tensor.dscalar('x')
f = theano.function([x], [5 * x],
                    mode=theano.compile.MonitorMode(
                        pre_func=inspect_inputs,
                        post_func=inspect_outputs))
f(3)
```

```
# The code will print the following:
#   0 Elemwise{mul,no_inplace}(TensorConstant{5.0}, x) input(s) value(s): [array(5.0), arr
```

When using these `inspect_inputs` and `inspect_outputs` functions with `MonitorMode`, you should see [potentially a lot of] printed output. Every `Apply` node will be printed out, along with its position in the graph, the arguments to the functions `perform` or `c_code` and the output it computed. Admittedly, this may be a huge amount of output to read through if you are using big tensors... but you can choose to add logic that would, for instance, print something out only if a certain kind of op were used, at a certain program position, or only if a particular value showed up in one of the inputs or outputs. A typical example is to detect when NaN values are added into computations, which can be achieved as follows:

```
import numpy

import theano

# This is the current suggested detect_nan implementation to
# show you how it work.  That way, you can modify it for your
# need.  If you want exactly this method, you can use
# ``theano.compile.monitormode.detect_nan`` that will always
# contain the current suggested version.

def detect_nan(i, node, fn):
    for output in fn.outputs:
        if (not isinstance(output[0], numpy.random.RandomState) and
            numpy.isnan(output[0]).any()):
            print '*** NaN detected ***'
            theano.printing.debugprint(node)
            print 'Inputs : %s' % [input[0] for input in fn.inputs]
            print 'Outputs: %s' % [output[0] for output in fn.outputs]
            break

x = theano.tensor.dscalar('x')
f = theano.function([x], [theano.tensor.log(x) * x],
                    mode=theano.compile.MonitorMode(
                        post_func=detect_nan))
f(0)  # log(0) * 0 = -inf * 0 = NaN

# The code above will print:
#   *** NaN detected ***
#   Elemwise{Composite{[mul(log(i0), i0)]}} [@A] ''
#    |x [@B]
#   Inputs : [array(0.0)]
#   Outputs: [array(nan)]
```

To help understand what is happening in your graph, you can disable the `local_elemwise_fusion` and all `inplace` optimizations. The first is a speed optimization that merges elemwise operations together. This makes it harder to know which particular elemwise causes the problem. The second optimization makes some ops' outputs overwrite their inputs. So, if an op creates a bad output, you will not be able to see the input that was overwriten in the `post_func` function. To disable those optimizations (with a Theano version after 0.6rc3), define the MonitorMode like this:

```
mode = theano.compile.MonitorMode(post_func=detect_nan).excluding(
    'local_elemwise_fusion', 'inplace')
f = theano.function([x], [theano.tensor.log(x) * x],
                    mode=mode)
```

Note: The Theano flags `optimizer_including`, `optimizer_excluding` and `optimizer_requiring` aren't used by the MonitorMode, they are used only by the `default` mode. You can't use the `default` mode with MonitorMode, as you need to define what you monitor.

To be sure all inputs of the node are available during the call to `post_func`, you must also disable the garbage collector. Otherwise, the execution of the node can garbage collect its inputs that aren't needed anymore by the Theano function. This can be done with the Theano flag:

```
allow_gc=False
```

How to Use pdb

In the majority of cases, you won't be executing from the interactive shell but from a set of Python scripts. In such cases, the use of the Python debugger can come in handy, especially as your models become more complex. Intermediate results don't necessarily have a clear name and you can get exceptions which are hard to decipher, due to the "compiled" nature of the functions.

Consider this example script ("ex.py"):

```
import theano
import numpy
import theano.tensor as T

a = T.dmatrix('a')
b = T.dmatrix('b')

f = theano.function([a, b], [a * b])

# matrices chosen so dimensions are unsuitable for multiplication
mat1 = numpy.arange(12).reshape((3, 4))
mat2 = numpy.arange(25).reshape((5, 5))

f(mat1, mat2)
```

This is actually so simple the debugging could be done easily, but it's for illustrative purposes. As the matrices can't be multiplied element-wise (unsuitable shapes), we get the following exception:

```
File "ex.py", line 14, in <module>
  f(mat1, mat2)
File "/u/username/Theano/theano/compile/function_module.py", line 451, in __call__
File "/u/username/Theano/theano/gof/link.py", line 271, in streamline_default_f
File "/u/username/Theano/theano/gof/link.py", line 267, in streamline_default_f
File "/u/username/Theano/theano/gof/cc.py", line 1049, in execute ValueError: ('Input dimer
```

The call stack contains some useful information to trace back the source of the error. There's the script where the compiled function was called – but if you're using (improperly parameterized) prebuilt modules,

the error might originate from ops in these modules, not this script. The last line tells us about the op that caused the exception. In this case it's a "mul" involving variables with names "a" and "b". But suppose we instead had an intermediate result to which we hadn't given a name.

After learning a few things about the graph structure in Theano, we can use the Python debugger to explore the graph, and then we can get runtime information about the error. Matrix dimensions, especially, are useful to pinpoint the source of the error. In the printout, there are also 2 of the 4 dimensions of the matrices involved, but for the sake of example say we'd need the other dimensions to pinpoint the error. First, we re-launch with the debugger module and run the program with "c":

```
python -m pdb ex.py
> /u/username/experiments/doctmp1/ex.py(1)<module>()
-> import theano
(Pdb) c
```

Then we get back the above error printout, but the interpreter breaks in that state. Useful commands here are

- "up" and "down" (to move up and down the call stack),
- "l" (to print code around the line in the current stack position),
- "p variable_name" (to print the string representation of 'variable_name'),
- "p dir(object_name)", using the Python dir() function to print the list of an object's members

Here, for example, I do "up", and a simple "l" shows me there's a local variable "node". This is the "node" from the computation graph, so by following the "node.inputs", "node.owner" and "node.outputs" links I can explore around the graph.

That graph is purely symbolic (no data, just symbols to manipulate it abstractly). To get information about the actual parameters, you explore the "thunk" objects, which bind the storage for the inputs (and outputs) with the function itself (a "thunk" is a concept related to closures). Here, to get the current node's first input's shape, you'd therefore do "p thunk.inputs[0][0].shape", which prints out "(3, 4)".

Dumping a Function to help debug

If you are reading this, there is high chance that you emailed our mailing list and we asked you to read this section. This section explain how to dump all the parameter passed to `theano.function()`. This is useful to help us reproduce a problem during compilation and it don't request you to make a self contained example.

For this to work, we need to be able to import the code for all Op in the graph. So if you create your own Op, we will need this code. Otherwise, we won't be able to unpickle it. We already have all the Ops from Theano and Pylearn2.

```
# Replace this line:
theano.function(...)
# with
theano.function_dump(filename, ...)
# Where filename is a string to a file that we will write to.
```

Then send us filename.

Dealing with NaNs

Having a model yielding NaNs or Infs is quite common if some of the tiny components in your model are not set properly. NaNs are hard to deal with because sometimes it is caused by a bug or error in the code, sometimes it's because of the numerical stability of your computational environment (library versions, etc.), and even, sometimes it relates to your algorithm. Here we try to outline common issues which cause the model to yield NaNs, as well as provide nails and hammers to diagnose it.

Check Superparameters and Weight Initialization

Most frequently, the cause would be that some of the hyperparameters, especially learning rates, are set incorrectly. A high learning rate can blow up your whole model into NaN outputs even within one epoch of training. So the first and easiest solution is try to lower it. Keep halving your learning rate until you start to get resonable output values.

Other hyperparameters may also play a role. For example, are your training algorithms involve regularization terms? If so, are their corresponding penalties set reasonably? Search a wider hyperparameter space with a few (one or two) training eopchs each to see if the NaNs could disappear.

Some models can be very sensitive to the initialization of weight vectors. If those weights are not initialized in a proper range, then it is not surprising that the model ends up with yielding NaNs.

Run in NanGuardMode, DebugMode, or MonitorMode

If adjusting hyperparameters doesn't work for you, you can still get help from Theano's NanGuardMode. change the mode of your theano function to NanGuardMode, and run them again. The NanGuardMode will monitor all input/output variables in each node, and raises an error if NaNs are detected. For how to use the `NanGuardMode`, please refer to *nanguardmode*.

DebugMode can also help. Run your code in DebugMode with flag mode=DebugMode, DebugMode.check_py=False. This will give you clue about which op is causing this problem, and then you can inspect into that op in more detail. For a detailed of using DebugMode, please refere to *debugmode*.

Theano's MonitorMode provides another helping hand. It can be used to step through the execution of a function. You can inspect the inputs and outputs of each node being executed when the function is called. For how to use that, please check *"How do I Step through a Compiled Function?"*.

Numerical Stability

After you have located the op which causes the problem, it may turn out that the NaNs yielded by that op are related to numerical issues. For example, :math: *1 / log(p(x) + 1)* may result in NaNs for those nodes who have learned to yield a low probability p(x) for some input x.

Algorithm Related

In the most difficult situations, you may go through the above steps and find nothing wrong. If the above methods fail to uncover the cause, there is a good chance that something is wrong with your algorithm. Go

back to the mathematics and find out if everything is derived correctly.

Profiling Theano function

Note: This method replace the old ProfileMode. Do not use ProfileMode anymore.

Besides checking for errors, another important task is to profile your code. For this, you can use Theano flags and/or parameters which are to be passed as an argument to `theano.function`.

The simplest way to profile Theano functions is to use the Theano flags described below. When the process exits, they will cause the information to be printed on stdout.

Using the ProfileMode is a three-step process.

Enabling the profiler is pretty easy. Just use the Theano flag `config.profile`.

To enable the memory profiler use the Theano flag: `config.profile_memory` in addition to `config.profile`.

To enable the profiling of Theano optimization phase, use the Theano flag: `config.profile_optimizer` in addition to `config.profile`.

You can use the Theano flags `profiling.n_apply`, `profiling.n_ops` and `profiling.min_memory_size` to modify the quantify of information printed.

The profiler will output one profile per Theano function and profile that is the sum of the printed profile. Each profile contains 4 sections: global info, class info, Ops info and Apply node info.

In the global section, the "Message" is the name of the Theano function. theano.function() has an optional parameter `name` that defaults to None. Change it to something else to help you profile many Theano functions. In that section, we also see the number of time the function was called (1) and the total time spent in all those calls. The time spent in Function.fn.__call__ and in thunks is useful to help understand Theano overhead.

Also, we see the time spent in the two parts of the compilation process: optimization(modify the graph to make it more stable/faster) and the linking (compile c code and make the Python callable returned by function).

The class, Ops and Apply nodes sections are the same information: information about the Apply node that ran. The Ops section takes the information from the Apply section and merge the Apply nodes that have exactly the same op. If two Apply nodes in the graph have two Ops that compare equal, they will be merged. Some Ops like Elemwise, will not compare equal, if their parameters differ (the scalar being executed). So the class section will merge more Apply nodes then the Ops section.

Here is an example output when we disable some Theano optimizations to give you a better idea of the difference between sections. With all optimizations enabled, there would be only one op left in the graph.

Note: To profile the peak memory usage on the GPU you need to do:

```
* In the file theano/sandbox/cuda/cuda_ndarray.cu, set the macro
  COMPUTE_GPU_MEM_USED to 1.
* Then call theano.sandbox.cuda.theano_allocated()
  It return a tuple with two ints. The first is the current GPU
```

```
  memory allocated by Theano. The second is the peak  GPU memory
  that was allocated by Theano.
```

Do not always enable this, as this slowdown memory allocation and free. As this slowdown the computation, this will affect speed profiling. So don't use both at the same time.

---

to run the example:

THEANO_FLAGS=optimizer_excluding=fusion:inplace,profile=True python doc/tutorial/profiling_example.py

The output:

```
Function profiling
==================
  Message: None
  Time in 1 calls to Function.__call__: 5.698204e-05s
  Time in Function.fn.__call__: 1.192093e-05s (20.921%)
  Time in thunks: 6.198883e-06s (10.879%)
  Total compile time: 3.642474e+00s
    Theano Optimizer time: 7.326508e-02s
       Theano validate time: 3.712177e-04s
    Theano Linker time (includes C, CUDA code generation/compiling): 9.584920e-01s

Class
---
<% time> <sum %> <apply time> <time per call> <type> <#call> <#apply> <Class name>
  100.0%   100.0%       0.000s       2.07e-06s     C        3        3   <class 'theano.ten
   ... (remaining 0 Classes account for   0.00%(0.00s) of the runtime)

Ops
---
<% time> <sum %> <apply time> <time per call> <type> <#call> <#apply> <Op name>
  65.4%    65.4%       0.000s       2.03e-06s     C        2        2   Elemwise{add,no_inp
  34.6%   100.0%       0.000s       2.15e-06s     C        1        1   Elemwise{mul,no_inp
   ... (remaining 0 Ops account for   0.00%(0.00s) of the runtime)

Apply
------
<% time> <sum %> <apply time> <time per call> <#call> <id> <Apply name>
  50.0%    50.0%       0.000s       3.10e-06s      1     0   Elemwise{add,no_inplace}(x, y)
  34.6%    84.6%       0.000s       2.15e-06s      1     2   Elemwise{mul,no_inplace}(Tenso
  15.4%   100.0%       0.000s       9.54e-07s      1     1   Elemwise{add,no_inplace}(Elemw
   ... (remaining 0 Apply instances account for 0.00%(0.00s) of the runtime)
```

Extending Theano

This tutorial covers how to extend Theano with novel ops. It mainly focuses on ops that offer a Python implementation, refers to *Extending Theano with a C Op* for C-based op. The first section of this tutorial introduces the Theano Graphs, as providing a novel Theano op requires a basic understanting of the Theano Graphs. It then proposes an overview of the most important methods that define an op.

As an illustration, this tutorial shows how to write a simple Python-based op which performs operations on Double. It also shows how to implement tests that ensure the proper working of an op.

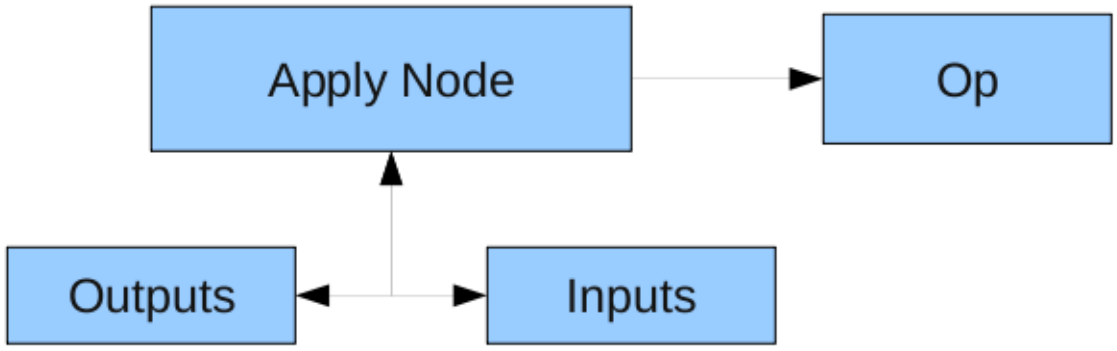
Note: This tutorial does not cover how to make an op that returns a view or modifies the values in its inputs. Thus, all ops created with the instructions described here MUST return newly allocated memory or reuse the memory provided in the parameter `output_storage` of the `perform()` function. See *Views and inplace operations* for an explanation on how to do this.

If your op returns a view or changes the value of its inputs without doing as prescribed in that page, Theano will run, but will return correct results for some graphs and wrong results for others.

It is recommended that you run your tests in DebugMode (Theano *flag* `mode=DebugMode`) since it verifies if your op behaves correctly in this regard.

Note: See the *Developer Start Guide* for information regarding the versioning framework, namely about *git* and *GitHub*, regarding the development workflow and how to make a quality contribution.

Theano Graphs



Theano represents symbolic mathematical computations as graphs. Those graphs are bi-partite graphs (graphs with 2 types of nodes), they are composed of interconnected *Apply* and *Variable* nodes. *Variable* nodes represent data in the graph, either inputs, outputs or intermediary values. As such, Inputs and Outputs of a graph are lists of Theano *Variable* nodes. *Apply* nodes perform computation on these variables to produce new variables. Each *Apply* node has a link to an instance of *Op* which describes the computation to perform. This tutorial details how to write such an Op instance. Please refers to *Graph Structures* for a more detailed explanation about the graph structure.

Op Structure

An op is any Python object which inherits from `gof.Op`. This section provides an overview of the methods you typically have to implement to make a new op. It does not provide extensive coverage of all the possibilities you may encounter or need. For that refer to *Op's contract*.

```
import theano

class MyOp(theano.Op):
    # Properties attribute
    __props__ = ()

    def make_node(self, *inputs):
        pass

    # Python implementation:
    def perform(self, node, inputs_storage, output_storage):
        pass

    # Other type of implementation
    # C implementation: [see theano web site for other functions]
    def c_code(...):
        # ...
        pass
    # Other implementations (pycuda, ...):
    def make_thunk(self, node, storage_map, _, _2):
        pass

    # optional:
    check_input = True

    def __init__(self, ...):
        pass

    def grad(self, inputs, g):
        pass

    def R_op(self, inputs, eval_points):
        pass

    def infer_shape(node, (i0_shapes, ...)):
        pass
```

An op has to implement some methods defined in the the interface of `gof.Op`. More specifically, it is mandatory for an op to define the method `make_node()` and one of the implementation methods, either `perform()`, `Op.c_code()` or `make_thunk()`.

`make_node()` method creates an Apply node representing the application of the op on the inputs provided. This method is reponsible for three things:

- it first checks that the input Variables types are compatible with the current op. If the op cannot be applied on the provided input types, it must raises an exception (such as `TypeError`).
- it operates on the Variables found in `*inputs` in Theano's symbolic language to infer the type of the symbolic output Variables. It creates output Variables of a suitable symbolic Type to serve as the outputs of this op's application.
- it creates an Apply instance with the input and output Variable, and return the Apply

instance.

`perform()` method defines the Python implementation of an op. It takes several arguments:

- `node` is a reference to an Apply node which was previously obtained via the `Op`'s `make_node()` method. It is typically not used in simple ops, but it contains symbolic information that could be required for complex ops.
- `inputs` is a list of references to data which can be operated on using non-symbolic statements, (i.e., statements in Python, Numpy).
- `output_storage` is a list of storage cells where the output is to be stored. There is one storage cell for each output of the op. The data put in `output_storage` must match the type of the symbolic output. It is forbidden to change the length of the list(s) contained in `output_storage`. A function Mode may allow `output_storage` elements to persist between evaluations, or it may reset `output_storage` cells to hold a value of `None`. It can also pre-allocate some memory for the op to use. This feature can allow `perform` to reuse memory between calls, for example. If there is something preallocated in the `output_storage`, it will be of the good dtype, but can have the wrong shape and have any stride pattern.

`perform()` method must be determined by the inputs. That is to say, when applied to identical inputs the method must return the same outputs.

`gof.Op` allows some other way to define the op implentation. For instance, it is possible to define `Op.c_code()` to provide a C-implementation to the op. Please refers to tutorial *Extending Theano with a C Op* for a description of `Op.c_code()` and other related c_methods. Note that an op can provide both Python and C implementation.

`make_thunk()` method is another alternative to `perform()`. It returns a thunk. A thunk is defined as a zero-arguments function which encapsulates the computation to be performed by an op on the arguments of its corresponding node. It takes several parameters:

- `node` is the Apply instance for which a thunk is requested,
- `storage_map` is a dict of lists which maps variables to a one-element lists holding the variable's current value. The one-element list acts as pointer to the value and allows sharing that "pointer" with other nodes and instances.
- `compute_map` is also a dict of lists. It maps variables to one-element lists holding booleans. If the value is 0 then the variable has not been computed and the value should not be considered valid. If the value is 1 the variable has been computed and the value is valid. If the value is 2 the variable has been garbage-collected and is no longer valid, but shouldn't be required anymore for this call. The returned function must ensure that it sets the computed variables as computed in the *compute_map*.

`make_thunk()` is useful if you want to generate code and compile it yourself. For example, this allows you to use PyCUDA to compile GPU code.

If `make_thunk()` is defined by an op, it will be used by Theano to obtain the op's implementation. `perform()` and `Op.c_code()` will be ignored.

Other methods can be optionally defined by the op.

The `__str__()` method provides a meaningful string representation of your op.

`__eq__()` and `__hash__()` define respectivelly equality between two ops and the hash of an op instance. They will be used by the optimization phase to merge nodes that are doing equivalent computations (same inputs, same operation). Two ops that are equal according `__eq__()` should return the same output when they are applied on the same inputs.

The `__props__` lists the properties that influence how the computation is performed (Ususally these are those that you set in `__init__()`). It must be a tuple. If you don't have any properties, then you should set this attribute to the emtpy tuple *()*.

`__props__` enables the automatic generation of appropriate `__eq__()` and `__hash__()`. Given the method `__eq__()`, automatically generated from `__props__`, two ops will be equal if they have the same values for all the properties listed in `__props__`. Given to the method `__hash__()` automatically generated from `__props__`, two ops will be have the same hash if they have the same values for all the properties listed in `__props__`. `__props__` will also generate a suitable `__str__()` for your op. This requires development version after September 1st, 2014 or version 0.7.

The `infer_shape()` method allows to infer the shape of the op output variables, without actually computing the outputs. It takes as input `node`, a reference to the op Apply node, and a list of Theano symbolic Varables (`i0_shape`, `i1_shape`, ...) which are the shape of the op input Variables. `infer_shape()` returns a list where each element is a tuple representing the shape of one output. This could be helpful if one only needs the shape of the output instead of the actual outputs, which can be useful, for instance, for optimization procedures.

The `grad()` method is required if you want to differentiate some cost whose expression includes your op. The gradient may be specified symbolically in this method. It takes two arguments `inputs` and `output_gradients` which are both lists of symbolic Theano Variables and those must be operated on using Theano's symbolic language. The grad method must return a list containing one Variable for each input. Each returned Variable represents the gradient with respect to that input computed based on the symbolic gradients with respect to each output. If the output is not differentiable with respect to an input then this method should be defined to return a variable of type NullType for that input. Likewise, if you have not implemented the grad computation for some input, you may return a variable of type NullType for that input. Please refer to `grad()` for a more detailed view.

The `R_op()` method is needed if you want `theano.tensor.Rop` to work with your op. This function implements the application of the R-operator on the function represented by your op. Let assume that function is f, with input x, applying the R-operator means computing the Jacobian of f and right-multiplying it by v, the evaluation point, namely: $\frac{\partial f}{\partial x} v$.

The optional boolean `check_input` attribute is used to specify if you want the types used in your op to check their inputs in their c_code. It can be used to speed up compilation, reduce overhead (particularly for scalars) and reduce the number of generated C files.

Op Example

```
import theano
```

```
class DoubleOp(theano.Op):
    __props__ = ()

    def make_node(self, x):
        # check that the theano version has support for __props__.
        # This next line looks like it has a typo,
        # but it's actually a way to detect the theano version
        # is sufficiently recent to support the use of __props__.
        assert hasattr(self, '_props'), "Your version of theano is too old to support __pro
        x = theano.tensor.as_tensor_variable(x)
        return theano.Apply(self, [x], [x.type()])

    def perform(self, node, inputs, output_storage):
        x = inputs[0]
        z = output_storage[0]
        z[0] = x * 2

    def infer_shape(self, node, i0_shapes):
        return i0_shapes

    def grad(self, inputs, output_grads):
        return [output_grads[0] * 2]

    def R_op(self, inputs, eval_points):
        # R_op can receive None as eval_points.
        # That mean there is no diferientiable path through that input
        # If this imply that you cannot compute some outputs,
        # return None for those.
        if eval_points[0] is None:
            return eval_points
        return self.grad(inputs, eval_points)
```

You can try it as follows:

```
x = theano.tensor.matrix()
f = theano.function([x], DoubleOp()(x))
import numpy
inp = numpy.random.rand(5, 4)
out = f(inp)
assert numpy.allclose(inp * 2, out)
print inp
print out
```

Example for properties of a Op

We can modify the previous piece of code in order to demonstrate the usage of the `__props__` attribute.

We create an Op that takes a variable `x` and returns `a*x+b`. We want to say that two such ops are equal when their values of `a` and `b` are equal.

```
import theano
```

```
class AXPBOp(theano.Op):
    """
    This creates an Op that takes x to a*x+b.
    """
    __props__ = ("a", "b")

    def __init__(self, a, b):
        self.a = a
        self.b = b
        super(AXPBOp, self).__init__()

    def make_node(self, x):
        # check that the theano version has support for __props__.
        assert hasattr(self, '_props'), "Your version of theano is too old to support __pro
        x = theano.tensor.as_tensor_variable(x)
        return theano.Apply(self, [x], [x.type()])

    def perform(self, node, inputs, output_storage):
        x = inputs[0]
        z = output_storage[0]
        z[0] = self.a * x + self.b

    def infer_shape(self, node, i0_shapes):
        return i0_shapes

    def grad(self, inputs, output_grads):
        return [a * output_grads[0] + b]
```

The use of `__props__` saves the user the trouble of implementing `__eq__()` and `__hash__()` manually. It also generates a default `__str__()` method that prints the attribute names and their values.

We can test this by running the following segment:

```
mult4plus5op = AXPBOp(4, 5)
another_mult4plus5op = AXPBOp(4, 5)
mult2plus3op = AXPBOp(2, 3)

assert mult4plus5op == another_mult4plus5op
assert mult4plus5op != mult2plus3op

x = theano.tensor.matrix()
f = theano.function([x], mult4plus5op(x))
g = theano.function([x], mult2plus3op(x))

import numpy
inp = numpy.random.rand(5, 4).astype(numpy.float32)
assert numpy.allclose(4 * inp + 5, f(inp))
assert numpy.allclose(2 * inp + 3, g(inp))
```

How To Test it

Theano has some functionalities to simplify testing. These help test the `infer_shape`, `grad` and `R_op` methods. Put the following code in a file and execute it with the `theano-nose` program.

Basic Tests Basic tests are done by you just by using the op and checking that it returns the right answer. If you detect an error, you must raise an *exception*. You can use the `assert` keyword to automatically raise an `AssertionError`.

```
from theano.tests import unittest_tools as utt
from theano import config
class test_Double(utt.InferShapeTester):
    def setUp(self):
        super(test_Double, self).setUp()
        self.op_class = DoubleOp
        self.op = DoubleOp()

    def test_basic(self):
        x = theano.tensor.matrix()
        f = theano.function([x], self.op(x))
        inp = numpy.asarray(numpy.random.rand(5, 4), dtype=config.floatX)
        out = f(inp)
        # Compare the result computed to the expected value.
        utt.assert_allclose(inp * 2, out)
```

We call `utt.assert_allclose(expected_value, value)` to compare NumPy ndarray.This raise an error message with more information. Also, the default tolerance can be changed with the Theano flags `config.tensor.cmp_sloppy` that take values in 0, 1 and 2. The defaul value do the most strict comparison, 1 and 2 make less strict comparison.

Testing the infer_shape When a class inherits from the `InferShapeTester` class, it gets the `self._compile_and_check` method that tests the op's `infer_shape` method. It tests that the op gets optimized out of the graph if only the shape of the output is needed and not the output itself. Additionally, it checks that the optimized graph computes the correct shape, by comparing it to the actual shape of the computed output.

`self._compile_and_check` compiles a Theano function. It takes as parameters the lists of input and output Theano variables, as would be provided to `theano.function`, and a list of real values to pass to the compiled function. It also takes the op class as a parameter in order to verify that no instance of it appears in the shape-optimized graph.

If there is an error, the function raises an exception. If you want to see it fail, you can implement an incorrect `infer_shape`.

When testing with input values with shapes that take the same value over different dimensions (for instance, a square matrix, or a tensor3 with shape (n, n, n), or (m, n, m)), it is not possible to detect if the output shape was computed correctly, or if some shapes with the same value have been mixed up. For instance, if the infer_shape uses the width of a matrix instead of its height, then testing with only square matrices will not detect the problem. This is why the `self._compile_and_check` method prints a warning in

such a case. If your op works only with such matrices, you can disable the warning with the `warn=False` parameter.

```
from theano.tests import unittest_tools as utt
from theano import config
class test_Double(utt.InferShapeTester):
    # [...] as previous tests.
    def test_infer_shape(self):
        x = theano.tensor.matrix()
        self._compile_and_check([x],  # theano.function inputs
                                [self.op(x)],  # theano.function outputs
                                # Always use not square matrix!
                                # inputs data
                                [numpy.asarray(numpy.random.rand(5, 4),
                                               dtype=config.floatX)],
                                # Op that should be removed from the graph.
                                self.op_class)
```

Testing the gradient The function *verify_grad* verifies the gradient of an op or Theano graph. It compares the analytic (symbolically computed) gradient and the numeric gradient (computed through the Finite Difference Method).

If there is an error, the function raises an exception. If you want to see it fail, you can implement an incorrect gradient (for instance, by removing the multiplication by 2).

```
def test_grad(self):
    theano.tests.unittest_tools.verify_grad(self.op,
                                            [numpy.random.rand(5, 7, 2)])
```

Testing the Rop The class `RopLop_checker` defines the functions `RopLop_checker.check_mat_rop_lop()`, `RopLop_checker.check_rop_lop()` and `RopLop_checker.check_nondiff_rop()`. These allow to test the implementation of the Rop method of a particular op.

For instance, to verify the Rop method of the DoubleOp, you can use this:

```
import numpy
import theano.tests
from theano.tests.test_rop import RopLop_checker
class test_DoubleRop(RopLop_checker):
    def setUp(self):
        super(test_DoubleRop, self).setUp()
    def test_double_rop(self):
        self.check_rop_lop(DoubleRop()(self.x), self.in_shape)
```

Testing GPU Ops Ops to be executed on the GPU should inherit from the `theano.sandbox.cuda.GpuOp` and not `theano.Op`. This allows Theano to distinguish them. Currently, we use this to test if the NVIDIA driver works correctly with our sum reduction code on the GPU.

Running Your Tests

To perform your tests, you may select either one of the three following methods:

theano-nose The method of choice to conduct tests is to run the file `theano-nose`. In a regular Theano installation, the latter will be on the operating system's path and directly accessible from any folder. Otherwise, it can be accessed in the `Theano/bin` folder. The following command lines may be used for the corresponding purposes:

- `theano-nose --theano`: Run every test found in Theano's path.
- `theano-nose folder_name`: Run every test found in the folder *folder_name*.
- `theano-nose test_file.py`: Run every test found in the file *test_file.py*.

The following are particularly useful for development purposes since they call for particular classes or even for particular tests:

- `theano-nose test_file.py:test_DoubleRop`: Run every test found inside the class *test_DoubleRop*.
- `theano-nose test_file.py:test_DoubleRop.test_double_op`: Run only the test *test_double_op* in the class *test_DoubleRop*.

Help with the use and functionalities of `theano-nose` may be obtained by running it with the command line parameter `--help (-h)`.

nosetests The command `nosetests` can also be used. Although it lacks the useful functionalities that `theano-nose` provides, `nosetests` can be called similarly to `theano-nose` from any folder in Python's path like so:

```
nosetests [suffix similar to the above].
```

More documentation on `nosetests` is available here: nosetests.

In-file One may also add a block of code similar to the following at the end of the file containing a specific test of interest and run the file. In this example, the test *test_DoubleRop* in the class *test_double_op* would be performed.

```
if __name__ == '__main__':
    t = test_DoubleRop("test_double_rop")
    t.setUp()
    t.test_double_rop()
```

We recommend that when we execute a file, we run all tests in that file. This can be done by adding this at the end of your test files:

```
if __name__ == '__main__':
     unittest.main()
```

Exercise

Run the code of the *DoubleOp* example above.

Modify and execute to compute: x * y.

Modify and execute the example to return two outputs: x + y and x - y.

You can omit the Rop functions. Try to implement the testing apparatus described above.

(Notice that Theano's current *elemwise fusion* optimization is only applicable to computations involving a single output. Hence, to gain efficiency over the basic solution that is asked here, the two operations would have to be jointly optimized explicitly in the code.)

as_op

as_op is a python decorator that converts a python function into a basic Theano op that will call the supplied function during execution.

This isn't the recommended way to build an op, but allows for a quick implementation.

It takes an optional `infer_shape()` parameter that must have this signature:

```
def infer_shape(node, input_shapes):
    # ...
    return output_shapes
```

- *input_shapes* and *output_shapes* are lists of tuples that represent the shape of the corresponding inputs/outputs.

Note: Not providing the *infer_shape* method prevents shape-related optimizations from working with this op. For example *your_op(inputs, ...).shape* will need the op to be executed just to get the shape.

Note: As no grad is defined, this means you won't be able to differentiate paths that include this op.

Note: It converts the Python function to a callable object that takes as inputs Theano variables that were declared.

---

as_op Example

```
import theano
import numpy
from theano.compile.ops import as_op

def infer_shape_numpy_dot(node, input_shapes):
    ashp, bshp = input_shapes
    return [ashp[:-1] + bshp[-1:]]

@as_op(itypes=[theano.tensor.fmatrix, theano.tensor.fmatrix],
       otypes=[theano.tensor.fmatrix], infer_shape=infer_shape_numpy_dot)
```

```
def numpy_dot(a, b):
   return numpy.dot(a, b)
```

You can try it as follows:

```
x = theano.tensor.fmatrix()
y = theano.tensor.fmatrix()
f = function([x, y], numpy_dot(x, y))
inp1 = numpy.random.rand(5, 4)
inp2 = numpy.random.rand(4, 7)
out = f(inp1, inp2)
```

Exercise Run the code of the *numpy_dot* example above.

Modify and execute to compute: numpy.add and numpy.subtract.

Modify and execute the example to return two outputs: x + y and x - y.

Random numbers in tests

Making tests errors more reproducible is a good practice. To make your tests more reproducible, you need a way to get the same random numbers. You can do this by seeding NumPy's random number generator.

For convenience, the classes InferShapeTester and RopLop_checker already do this for you. If you implement your own `setUp` function, don't forget to call the parent `setUp` function.

For more details see *Using Random Values in Test Cases*.

`Solution`

Documentation

See *Documentation Documentation AKA Meta-Documentation*, for some information on how to generate the documentation.

Here is an example how to add docstring to a class.

```
import theano

class DoubleOp(theano.Op):
""" Double each element of a tensor.

:param x: input tensor.

:return: a tensor of the same shape and dtype as the input with all
    values doubled.

:note:
    this is a test note

:seealso:
```

```
    You can use the elemwise op to replace this example.
    Just execute `x * 2` with x being a Theano variable.

.. versionadded:: 0.6
"""
```

This is how it will show up for files that we auto-list in the library documentation:

class `theano.misc.doubleop.`**`DoubleOp`**(*use_c_code='/usr/bin/g++'*)

Double each element of a tensor.

Parameters **x** – input tensor.

Returns a tensor of the same shape and dtype as the input with all values doubled.

Note this is a test note

Seealso You can use the elemwise op to replace this example. Just execute *x * 2* with x being a Theano variable.

New in version 0.6.

Final Note

A more extensive discussion of this section's content may be found in the advanced tutorial *Extending Theano*.

The section *Other ops* includes more instructions for the following specific cases:

- *Scalar/Elemwise/Reduction Ops*
- *SciPy Ops*
- *Sparse Ops*
- *Random ops*
- *OpenMP Ops*
- *Numba Ops*

Extending Theano with a C Op

This tutorial covers how to extend Theano with an op that offers a C implementation. It does not cover ops that run on a GPU but it does introduce many elements and concepts which are relevant for GPU ops. This tutorial is aimed at individuals who already know how to extend Theano (see tutorial *Extending Theano*) by adding a new op with a Python implementation and will only cover the additional knowledge required to also produce ops with C implementations.

Providing a Theano op with a C implementation requires to interact with Python's C-API and Numpy's C-API. Thus, the first step of this tutorial is to introduce both and highlight their features which are most relevant to the task of implementing a C op. This tutorial then introduces the most important methods that the op needs to implement in order to provide a usable C implementation. Finally, it shows how to combine

these elements to write a simple C op for performing the simple task of multiplying every element in a vector by a scalar.

Python C-API

Python provides a C-API to allows the manipulation of python objects from C code. In this API, all variables that represent Python objects are of type `PyObject *`. All objects have a pointer to their type object and a reference count field (that is shared with the python side). Most python methods have an equivalent C function that can be called on the `PyObject *` pointer.

As such, manipulating a PyObject instance is often straight-forward but it is important to properly manage its reference count. Failing to do so can lead to undesired behavior in the C code.

Reference counting Reference counting is a mechanism for keeping track, for an object, of the number of references to it held by other entities. This mechanism is often used for purposes of garbage collecting because it allows to easily see if an object is still being used by other entities. When the reference count for an object drops to 0, it means it is not used by anyone any longer and can be safely deleted.

PyObjects implement reference counting and the Python C-API defines a number of macros to help manage those reference counts. The definition of these macros can be found here : Python C-API Reference Counting. Listed below are the two macros most often used in Theano C ops.

`void Py_XINCREF(PyObject *o)`
Increments the reference count of object `o`. Without effect if the object is NULL.

`void Py_XDECREF(PyObject *o)`
Decrements the reference count of object `o`. If the reference count reaches 0, it will trigger a call of the object's deallocation function. Without effect if the object is NULL.

The general principle, in the reference counting paradigm, is that the owner of a reference to an object is responsible for disposing properly of it. This can be done by decrementing the reference count once the reference is no longer used or by transfering ownership; passing on the reference to a new owner which becomes responsible for it.

Some functions return "borrowed references"; this means that they return a reference to an object **without** transfering ownership of the reference to the caller of the function. This means that if you call a function which returns a borrowed reference, you do not have the burden of properly disposing of that reference. You should **not** call Py_XDECREF() on a borrowed reference.

Correctly managing the reference counts is important as failing to do so can lead to issues ranging from memory leaks to segmentation faults.

NumPy C-API

The NumPy library provides a C-API to allow users to create, access and manipulate NumPy arrays from within their own C routines. NumPy's ndarrays are used extensively inside Theano and so extending Theano with a C op will require interaction with the NumPy C-API.

This sections covers the API's elements that are often required to write code for a Theano C op. The full documentation for the API can be found here : NumPy C-API.

NumPy data types To allow portability between platforms, the NumPy C-API defines its own data types which should be used whenever you are manipulating a NumPy array's internal data. The data types most commonly used to implement C ops are the following : `npy_int{8,16,32,64}`, `npy_uint{8,16,32,64}` and `npy_float{32,64}`.

You should use these data types when manipulating a NumPy array's internal data instead of C primitives because the size of the memory representation for C primitives can vary between platforms. For instance, a C `long` can be represented in memory with 4 bytes but it can also be represented with 8. On the other hand, the in-memory size of NumPy data types remains constant across platforms. Using them will make your code simpler and more portable.

The full list of defined data types can be found here : NumPy C-API data types.

NumPy ndarrays In the NumPy C-API, NumPy arrays are represented as instances of the PyArrayObject class which is a descendant of the PyObject class. This means that, as for any other Python object that you manipulate from C code, you need to appropriatedly manage the reference counts of PyArrayObject instances.

Unlike in a standard multidimensionnal C array, a NumPy array's internal data representation does not have to occupy a continuous region in memory. In fact, it can be C-contiguous, F-contiguous or non-contiguous. C-contiguous means that the data is not only contiguous in memory but also that it is organized such that the index of the latest dimension changes the fastest. If the following array

```
x = [[1, 2, 3],
     [4, 5, 6]]
```

is C-contiguous, it means that, in memory, the six values contained in the array `x` are stored in the order `[1, 2, 3, 4, 5, 6]` (the first value is `x[0,0]`, the second value is `x[0,1]`, the third value is `x[0,2]`, the, fourth value is `x[1,0]`, etc). F-contiguous (or Fortran Contiguous) also means that the data is contiguous but that it is organized such that the index of the latest dimension changes the slowest. If the array `x` is F-contiguous, it means that, in memory, the values appear in the order `[1, 4, 2, 5, 3, 6]` (the first value is `x[0,0]`, the second value is `x[1,0]`, the third value is `x[0,1]`, etc).

Finally, the internal data can be non-contiguous. In this case, it occupies a non-contiguous region in memory but it is still stored in an organized fashion : the distance between the element `x[i,j]` and the element `x[i+1,j]` of the array is constant over all valid values of `i` and `j`, just as the distance between the element `x[i,j]` and the element `x[i,j+1]` of the array is constant over all valid values of `i` and `j`. This distance between consecutive elements of an array over a given dimension, is called the stride of that dimension.

Accessing NumPy ndarrays' data and properties The following macros serve to access various attributes of NumPy ndarrays.

`void* PyArray_DATA(PyArrayObject* arr)`
Returns a pointer to the first element of the array's data. The returned pointer must be cast to a pointer of the proper Numpy C-API data type before use.

int PyArray_NDIM(PyArrayObject* arr)
Returns the number of dimensions in the the array pointed by `arr`

npy_intp* PyArray_DIMS(PyArrayObject* arr)
Returns a pointer on the first element of `arr`'s internal array describing its dimensions. This internal array contains as many elements as the array `arr` has dimensions.

The macro `PyArray_SHAPE()` is a synonym of `PyArray_DIMS()` : it has the same effect and is used in an identical way.

npy_intp* PyArray_STRIDES(PyArrayObject* arr)
Returns a pointer on the first element of `arr`'s internal array describing the stride for each of its dimension. This array has as many elements as the number of dimensions in `arr`. In this array, the strides are expressed in number of bytes.

PyArray_Descr* PyArray_DESCR(PyArrayObject* arr)
Returns a reference to the object representing the dtype of the array.

The macro `PyArray_DTYPE()` is a synonym of the `PyArray_DESCR()` : it has the same effect and is used in an identical way.

Note This is a borrowed reference so you do not need to decrement its reference count once you are done with it.

int PyArray_TYPE(PyArrayObject* arr)
Returns the typenumber for the elements of the array. Like the dtype, the typenumber is a descriptor for the type of the data in the array. However, the two are not synonyms and, as such, cannot be used in place of the other.

npy_intp PyArray_SIZE(PyArrayObject* arr)
Returns to total number of elements in the array

bool PyArray_CHKFLAGS(PyArrayObject* arr, flags)
Returns true if the array has the specified flags. The variable flag should either be a NumPy array flag or an integer obtained by applying bitwise or to an ensemble of flags.

The flags that can be used in with this macro are : NPY_ARRAY_C_CONTIGUOUS, NPY_ARRAY_F_CONTIGUOUS, NPY_ARRAY_OWNDATA, NPY_ARRAY_ALIGNED, NPY_ARRAY_WRITEABLE, NPY_ARRAY_UPDATEIFCOPY.

Creating NumPy ndarrays The following functions allow the creation and copy of NumPy arrays :

PyObject* PyArray_EMPTY(int nd, npy_intp* dims, typenum dtype, int fortran)
Constructs a new ndarray with the number of dimensions specified by `nd`, shape specified by `dims` and data type specified by `dtype`. If `fortran` is equal to 0, the data is organized in a C-contiguous layout, otherwise it is organized in a F-contiguous layout. The array elements are not initialized in any way.

The function `PyArray_Empty()` performs the same function as the macro `PyArray_EMPTY()` but the data type is given as a pointer to a `PyArray_Descr` object instead of a `typenum`.

PyObject* PyArray_ZEROS(int nd, npy_intp* dims, typenum dtype,

int fortran)
Constructs a new ndarray with the number of dimensions specified by `nd`, shape specified by `dims` and data type specified by `dtype`. If `fortran` is equal to 0, the data is organized in a C-contiguous layout, otherwise it is organized in a F-contiguous layout. Every element in the array is initialized to 0.

The function `PyArray_Zeros()` performs the same function as the macro `PyArray_ZEROS()` but the data type is given as a pointer to a `PyArray_Descr` object instead of a `typenum`.

PyArrayObject* PyArray_GETCONTIGUOUS(PyObject* op)
Returns a C-contiguous and well-behaved copy of the array op. If op is already C-contiguous and well-behaved, this function simply returns a new reference to op.

Methods the C Op needs to define

There is a key difference between an op defining a Python implementation for its computation and defining a C implementation. In the case of a Python implementation, the op defines a function `perform()` which executes the required Python code to realize the op. In the case of a C implementation, however, the op does **not** define a function that will execute the C code; it instead defines functions that will **return** the C code to the caller.

This is because calling C code from Python code comes with a significant overhead. If every op was responsible for executing its own C code, every time a Theano function was called, this overhead would occur as many times as the number of ops with C implementations in the function's computational graph.

To maximize performance, Theano instead requires the C ops to simply return the code needed for their execution and takes upon itself the task of organizing, linking and compiling the code from the various ops. Through this, Theano is able to minimize the number of times C code is called from Python code.

The following is a very simple example to illustrate how it's possible to obtain performance gains with this process. Suppose you need to execute, from Python code, 10 different ops, each one having a C implementation. If each op was responsible for executing its own C code, the overhead of calling C code from Python code would occur 10 times. Consider now the case where the ops instead return the C code for their execution. You could get the C code from each op and then define your own C module that would call the C code from each op in succession. In this case, the overhead would only occur once; when calling your custom module itself.

Moreover, the fact that Theano itself takes care of compiling the C code, instead of the individual ops, allows Theano to easily cache the compiled C code. This allows for faster compilation times.

See *Implementing the arithmetic Ops in C* for the full documentation of the various methods of the class Op that are related to the C implementation. Of particular interest are:

- The methods `Op.c_libraries()` and `Op.c_lib_dirs()` to allow your op to use external libraries.
- The method `Op.c_code_cleanup()` to specify how the op should clean up what it has allocated during its execution.
- The methods `Op.c_init_code()` and `Op.c_init_code_apply()` to specify code that should be executed once when the module is initialized, before anything else is executed.

- The methods `Op.c_compile_args()` and `Op.c_no_compile_args()` to specify requirements regarding how the op's C code should be compiled.

This section describes the methods `Op.c_code()`, `Op.c_support_code()`, `Op.c_support_code_apply()` and `Op.c_code_cache_version()` because they are the ones that are most commonly used.

c_code (*node*, *name*, *input_names*, *output_names*, *sub*)

This method returns a string containing the C code to perform the computation required by this op.

The `node` argument is an *Apply* node representing an application of the current Op on a list of inputs, producing a list of outputs.

`input_names` is a sequence of strings which contains as many strings as the op has inputs. Each string contains the name of the C variable to which the corresponding input has been assigned. For example, the name of the C variable representing the first input of the op is given by `input_names[0]`. You should therefore use this name in your C code to interact with that variable. `output_names` is used identically to `input_names`, but for the op's outputs.

Finally, `sub` is a dictionary of extras parameters to the c_code method. Among other things, it contains `sub['fail']` which is a string of C code that you should include in your C code (after ensuring that a Python exception is set) if it needs to raise an exception. Ex:

```
c_code = """
    PyErr_Format(PyExc_ValueError, "X does not have the right value");
    %(fail)s;
""" % {'fail' : sub['fail']}
```

to raise a ValueError Python exception with the specified message. The function `PyErr_Format()` supports string formatting so it is possible to tailor the error message to the specifics of the error that occured. If `PyErr_Format()` is called with more than two arguments, the subsequent arguments are used to format the error message with the same behavior as the function PyString_FromFormat(). The `%` characters in the format characters need to be escaped since the C code itself is defined in a string which undergoes string formatting.

```
c_code = """
    PyErr_Format(PyExc_ValueError,
                 "X==%%i but it should be greater than 0", X);
    %(fail)s;
""" % {'fail' : sub['fail']}
```

Note Your C code should not return the output of the computation but rather put the results in the C variables whose names are contained in the `output_names`.

c_support_code()

Returns a string containing some support C code for this op. This code will be included at the global scope level and can be used to define functions and structs that will be used by every apply of this op.

c_support_code_apply (*node*, *name*)

Returns a string containing some support C code for this op. This code will be included at the global scope level and can be used to define functions and structs that will be used by this op. The difference between this method and `c_support_code()` is that the C code

specified in `c_support_code_apply()` should be specific to each apply of the Op, while `c_support_code()` is for support code that is not specific to each apply.

Both `c_support_code()` and `c_support_code_apply ()` are necessary because a Theano op can be used more than once in a given Theano function. For example, an op that adds two matrices could be used at some point in the Theano function to add matrices of integers and, at another point, to add matrices of doubles. Because the dtype of the inputs and outputs can change between different applies of the op, any support code that relies on a certain dtype is specific to a given apply of the op and should therefore be defined in `c_support_code_apply()`.

`c_code_cache_version``()`

Returns a tuple of integers representing the version of the C code in this op. Ex : (1, 4, 0) for version 1.4.0

This tuple is used by Theano to cache the compiled C code for this op. As such, the return value **MUST BE CHANGED** every time the C code is altered or else Theano will disregard the change in the code and simply load a previous version of the op from the cache. If you want to avoid caching of the C code of this op, return an empty tuple or do not implement this method.

Note Theano can handle tuples of any hashable objects as return values for this function but, for greater readability and easier management, this function should return a tuple of integers as previously described.

Simple C Op example

In this section, we put together the concepts that were covered in this tutorial to generate an op which multiplies every element in a vector by a scalar and returns the resulting vector. This is intended to be a simple example so the methods `c_support_code()` and `c_support_code_apply()` are not used because they are not required.

In the C code below notice how the reference count on the output variable is managed. Also take note of how the new variables required for the op's computation are declared in a new scope to avoid cross-initialization errors.

Also, in the C code, it is very important to properly validate the inputs and outputs storage. Theano guarantees that the inputs exist and have the right number of dimensions but it does not guarantee their exact shape. For instance, if an op computes the sum of two vectors, it needs to validate that its two inputs have the same shape. In our case, we do not need to validate the exact shapes of the inputs because we don't have a need that they match in any way.

For the outputs, things are a little bit more subtle. Theano does not guarantee that they have been allocated but it does guarantee that, if they have been allocated, they have the right number of dimension. Again, Theano offers no guarantee on the exact shapes. This means that, in our example, we need to validate that the output storage has been allocated and has the same shape as our vector input. If it is not the case, we allocate a new output storage with the right shape and number of dimensions.

```
import numpy
import theano
from theano import gof
import theano.tensor as T
```

```
class VectorTimesScalar(gof.Op):
    __props__ = ()

    def make_node(self, x, y):
        # Validate the inputs' type
        if x.type.ndim != 1:
            raise TypeError('x must be a 1-d vector')
        if y.type.ndim != 0:
            raise TypeError('y must be a scalar')

        # Create an output variable of the same type as x
        output_var = x.type()

        return gof.Apply(self, [x, y], [output_var])

    def c_code_cache_version(self):
        return (1, 0)

    def c_code(self, node, name, inp, out, sub):
        x, y = inp
        z, = out

        # Extract the dtypes of the inputs and outputs storage to
        # be able to declare pointers for those dtypes in the C
        # code.
        dtype_x = node.inputs[0].dtype
        dtype_y = node.inputs[1].dtype
        dtype_z = node.outputs[0].dtype

        itemsize_x = numpy.dtype(dtype_x).itemsize
        itemsize_z = numpy.dtype(dtype_z).itemsize

        fail = sub['fail']

        c_code = """
        // Validate that the output storage exists and has the same
        // dimension as x.
        if (NULL == %(z)s ||
            PyArray_DIMS(%(x)s)[0] != PyArray_DIMS(%(z)s)[0])
        {
            /* Reference received to invalid output variable.
            Decrease received reference's ref count and allocate new
            output variable */
            Py_XDECREF(%(z)s);
            %(z)s = (PyArrayObject*)PyArray_EMPTY(1,
                                                  PyArray_DIMS(%(x)s),
                                                  PyArray_TYPE(%(x)s),
                                                  0);

            if (!%(z)s) {
                %(fail)s;
            }
        }
```

```
        // Perform the vector multiplication by a scalar
        {
            /* The declaration of the following variables is done in a new
            scope to prevent cross initialization errors */
            npy_%(dtype_x)s* x_data_ptr =
                            (npy_%(dtype_x)s*)PyArray_DATA(%(x)s);
            npy_%(dtype_z)s* z_data_ptr =
                            (npy_%(dtype_z)s*)PyArray_DATA(%(z)s);
            npy_%(dtype_y)s y_value =
                            ((npy_%(dtype_y)s*)PyArray_DATA(%(y)s))[0];
            int x_stride = PyArray_STRIDES(%(x)s)[0] / %(itemsize_x)s;
            int z_stride = PyArray_STRIDES(%(z)s)[0] / %(itemsize_z)s;
            int x_dim = PyArray_DIMS(%(x)s)[0];

            for(int i=0; i < x_dim; i++)
            {
                z_data_ptr[i * z_stride] = (x_data_ptr[i * x_stride] *
                                            y_value);
            }
        }
        """

        return c_code % locals()
```

More complex C Op example

This section introduces a new example, slightly more complex than the previous one, with an op to perform an element-wise multiplication between the elements of two vectors. This new example differs from the previous one in its use of the methods `c_support_code()` and `c_support_code_apply()` (it does not *need* to use them but it does so to explain their use) and its capacity to support inputs of different dtypes.

Recall the method `c_support_code()` is meant to produce code that will be used for every apply of the op. This means that the C code in this method must be valid in every setting your op supports. If the op is meant to supports inputs of various dtypes, the C code in this method should be generic enough to work with every supported dtype. If the op operates on inputs that can be vectors or matrices, the C code in this method should be able to accomodate both kinds of inputs.

In our example, the method `c_support_code()` is used to declare a C function to validate that two vectors have the same shape. Because our op only supports vectors as inputs, this function is allowed to rely on its inputs being vectors. However, our op should support multiple dtypes so this function cannot rely on a specific dtype in its inputs.

The method `c_support_code_apply()`, on the other hand, is allowed to depend on the inputs to the op because it is apply-specific. Therefore, we use it to define a function to perform the multiplication between two vectors. Variables or functions defined in the method `c_support_code_apply()` will be included at the global scale for every apply of the Op. Because of this, the names of those variables and functions should include the name of the op, like in the example. Otherwise, using the op twice in the same graph will give rise to conflicts as some elements will be declared more than once.

The last interesting difference occurs in the `c_code()` method. Because the dtype of the output is variable

and not guaranteed to be the same as any of the inputs (because of the upcast in the method `make_node()`), the typenum of the output has to be obtained in the Python code and then included in the C code.

```
class VectorTimesVector(gof.Op):
    __props__ = ()

    def make_node(self, x, y):
        # Validate the inputs' type
        if x.type.ndim != 1:
            raise TypeError('x must be a 1-d vector')
        if y.type.ndim != 1:
            raise TypeError('y must be a 1-d vector')

        # Create an output variable of the same type as x
        output_var = theano.tensor.TensorType(
                        dtype=theano.scalar.upcast(x.dtype, y.dtype),
                        broadcastable=[False])()

        return gof.Apply(self, [x, y], [output_var])

    def c_code_cache_version(self):
        return (1, 0, 2)

    def c_support_code(self):
        c_support_code = """
        bool vector_same_shape(PyArrayObject* arr1,
            PyArrayObject* arr2)
        {
            return (PyArray_DIMS(arr1)[0] == PyArray_DIMS(arr2)[0]);
        }
        """

        return c_support_code

    def c_support_code_apply(self, node, name):
        dtype_x = node.inputs[0].dtype
        dtype_y = node.inputs[1].dtype
        dtype_z = node.outputs[0].dtype

        c_support_code = """
        void vector_elemwise_mult_%(name)s(npy_%(dtype_x)s* x_ptr,
            int x_str, npy_%(dtype_y)s* y_ptr, int y_str,
            npy_%(dtype_z)s* z_ptr, int z_str, int nbElements)
        {
            for (int i=0; i < nbElements; i++){
                z_ptr[i * z_str] = x_ptr[i * x_str] * y_ptr[i * y_str];
            }
        }
        """

        return c_support_code % locals()

    def c_code(self, node, name, inp, out, sub):
        x, y = inp
```

```
        z, = out

        dtype_x = node.inputs[0].dtype
        dtype_y = node.inputs[1].dtype
        dtype_z = node.outputs[0].dtype

        itemsize_x = numpy.dtype(dtype_x).itemsize
        itemsize_y = numpy.dtype(dtype_y).itemsize
        itemsize_z = numpy.dtype(dtype_z).itemsize

        typenum_z = numpy.dtype(dtype_z).num

        fail = sub['fail']

        c_code = """
        // Validate that the inputs have the same shape
        if ( !vector_same_shape(%(x)s, %(y)s))
        {
            PyErr_Format(PyExc_ValueError, "Shape mismatch : "
                         "x.shape[0] and y.shape[0] should match but "
                         "x.shape[0] == %%i and y.shape[0] == %%i",
                         PyArray_DIMS(%(x)s)[0], PyArray_DIMS(%(y)s)[0]);
            %(fail)s;
        }

        // Validate that the output storage exists and has the same
        // dimension as x.
        if (NULL == %(z)s || !(vector_same_shape(%(x)s, %(z)s)))
        {
            /* Reference received to invalid output variable.
            Decrease received reference's ref count and allocate new
            output variable */
            Py_XDECREF(%(z)s);
            %(z)s = (PyArrayObject*)PyArray_EMPTY(1,
                                                  PyArray_DIMS(%(x)s),
                                                  %(typenum_z)s,
                                                  0);

            if (!%(z)s) {
                %(fail)s;
            }
        }

        // Perform the vector elemwise multiplication
        vector_elemwise_mult_%(name)s(
                                (npy_%(dtype_x)s*)PyArray_DATA(%(x)s),
                                PyArray_STRIDES(%(x)s)[0] / %(itemsize_x)s,
                                (npy_%(dtype_y)s*)PyArray_DATA(%(y)s),
                                PyArray_STRIDES(%(y)s)[0] / %(itemsize_y)s,
                                (npy_%(dtype_z)s*)PyArray_DATA(%(z)s),
                                PyArray_STRIDES(%(z)s)[0] / %(itemsize_z)s,
                                PyArray_DIMS(%(x)s)[0]);
        """
```

```
        return c_code % locals()
```

Alternate way of defining C Ops

The two previous examples have covered the standard way of implementing C Ops in Theano by inheriting from the class `Op`. This process is mostly simple but it still involves defining many methods as well as mixing, in the same file, both Python and C code which tends to make the result less readable.

To help with this, Theano defines a class, `COp`, from which new C ops can inherit. The class `COp` aims to simplify the process of implementing C ops by doing the following :

- It allows you to define the C implementation of your op in a distinct C code file. This makes it easier to keep your Python and C code readable and well indented.
- It can automatically handle all the methods that return C code, in addition to `Op.c_code_cache_version()` based on the provided external C implementation.

To illustrate how much simpler the class `COp` makes the process of defining a new op with a C implementation, let's revisit the second example of this tutorial, the `VectorTimesVector` op. In that example, we implemented an op to perform the task of element-wise vector-vector multiplication. The two following blocks of code illustrate what the op would look like if it was implemented using the `COp` class.

The new op is defined inside a Python file with the following code :

```
import theano
from theano import gof

class VectorTimesVector(gof.COp):
    __props__ = ()

    func_file = "./vectorTimesVector.c"
    func_name = "APPLY_SPECIFIC(vector_times_vector)"

    def __init__(self):
        super(VectorTimesVector, self).__init__(self.func_file,
                                                self.func_name)

    def make_node(self, x, y):
        # Validate the inputs' type
        if x.type.ndim != 1:
            raise TypeError('x must be a 1-d vector')
        if y.type.ndim != 1:
            raise TypeError('y must be a 1-d vector')

        # Create an output variable of the same type as x
        output_var = theano.tensor.TensorType(
                        dtype=theano.scalar.upcast(x.dtype, y.dtype),
                        broadcastable=[False])()

        return gof.Apply(self, [x, y], [output_var])
```

And the following is the C implementation of the op, defined in an external C file named vectorTimesVector.c :

```
#section support_code

// Support code function
bool vector_same_shape(PyArrayObject* arr1, PyArrayObject* arr2)
{
    return (PyArray_DIMS(arr1)[0] == PyArray_DIMS(arr2)[0]);
}


#section support_code_apply

// Apply-specific support function
void APPLY_SPECIFIC(vector_elemwise_mult)(
    DTYPE_INPUT_0* x_ptr, int x_str,
    DTYPE_INPUT_1* y_ptr, int y_str,
    DTYPE_OUTPUT_0* z_ptr, int z_str, int nbElements)
{
    for (int i=0; i < nbElements; i++){
        z_ptr[i * z_str] = x_ptr[i * x_str] * y_ptr[i * y_str];
    }
}

// Apply-specific main function
int APPLY_SPECIFIC(vector_times_vector)(PyArrayObject* input0,
                                        PyArrayObject* input1,
                                        PyArrayObject** output0)
{
    // Validate that the inputs have the same shape
    if ( !vector_same_shape(input0, input1))
    {
        PyErr_Format(PyExc_ValueError, "Shape mismatch : "
                     "input0.shape[0] and input1.shape[0] should "
                     "match but x.shape[0] == %i and "
                     "y.shape[0] == %i",
                     PyArray_DIMS(input0)[0], PyArray_DIMS(input1)[0]);
        return 1;
    }

    // Validate that the output storage exists and has the same
    // dimension as x.
    if (NULL == *output0 || !(vector_same_shape(input0, *output0)))
    {
        /* Reference received to invalid output variable.
        Decrease received reference's ref count and allocate new
        output variable */
        Py_XDECREF(*output0);
        *output0 = (PyArrayObject*)PyArray_EMPTY(1,
                                                 PyArray_DIMS(input0),
                                                 TYPENUM_OUTPUT_0,
                                                 0);
```

```
        if (!*output0) {
            PyErr_Format(PyExc_ValueError,
                         "Could not allocate output storage");
            return 1;
        }
    }

    // Perform the actual vector-vector multiplication
    APPLY_SPECIFIC(vector_elemwise_mult)(
                            (DTYPE_INPUT_0*)PyArray_DATA(input0),
                            PyArray_STRIDES(input0)[0] / ITEMSIZE_INPUT_0,
                            (DTYPE_INPUT_1*)PyArray_DATA(input1),
                            PyArray_STRIDES(input1)[0] / ITEMSIZE_INPUT_1,
                            (DTYPE_OUTPUT_0*)PyArray_DATA(*output0),
                            PyArray_STRIDES(*output0)[0] / ITEMSIZE_OUTPUT_0,
                            PyArray_DIMS(input0)[0]);

    return 0;
}
```

As you can see from this example, the Python and C implementations are nicely decoupled which makes them much more readable than when they were intertwined in the same file and the C code contained string formatting markers.

Now that we have motivated the COp class, we can have a more precise look at what it does for us. For this, we go through the various elements that make up this new version of the VectorTimesVector op :

- Parent class : instead of inheriting from the class `Op`, VectorTimesVector inherits from the class `COp`.
- Constructor : in our new op, the `__init__()` method has an important use; to inform the constructor of the `COp` class of the location, on the filesystem of the C implementation of this op. To do this, it gives a list of file paths containing the C code for this op. To auto-generate the c_code method with a function call you can specify the function name as the second parameter. The paths should be given as a relative path from the folder where the descendant of the `COp` class is defined.
- `make_node()` : the `make_node()` method is absolutely identical to the one in our old example. Using the `COp` class doesn't change anything here.
- External C code : the external C code implements the various functions associated with the op. Writing this C code involves a few subtleties which deserve their own respective sections.

Main function If you pass a function name to the `__init__()` method of the `COp` class, it must respect the following constraints:

- It must return an int. The value of that int indicates whether the op could perform its task or not. A value of 0 indicates success while any non-zero value will interrupt the execution of the Theano function. When returning non-zero the function must set a python exception indicating the details of the problem.
- It must receive one argument for each input to the op followed by one pointer to an argument for each output of the op. The types for the argument is dependant on the Types (that is theano Types) of your inputs and outputs.

For example, the main C function of an op that takes two TensorTypes (which has `PyArrayObject *` as its C type) as inputs and returns both their sum and the difference between them would have four parameters (two for the op's inputs and two for its outputs) and it's signature would look something like this :

```
int sumAndDiffOfScalars(PyArrayObject* in0, PyArrayObject* in1,
                        PyArrayObject** out0, PyArrayObject** out1)
```

Macros For certain section tags, your C code can benefit from a number of pre-defined macros. These section tags have no macros: `init_code`, `support_code`. All other tags will have the support macros discussed below.

- `APPLY_SPECIFIC(str)` which will automatically append a name unique to the *Apply node that applies the Op at the end of the provided ''str'*. The use of this macro is discussed futher below.

For every input which has a `dtype` attribute (this means Tensors, and equivalent types on GPU), the following macros will be defined unless your Op class has an `Op.check_input` attribute defined to False. In these descrptions 'i' refers to the position (indexed from 0) in the input array.

- `DTYPE_INPUT_{i}` : NumPy dtype of the data in the array. This is the variable type corresponding to the NumPy dtype, not the string representation of the NumPy dtype. For instance, if the op's first input is a float32 ndarray, then the macro `DTYPE_INPUT_0` corresponds to `npy_float32` and can directly be used to declare a new variable of the same dtype as the data in the array :

  ```
  DTYPE_INPUT_0 myVar = someValue;
  ```

- `TYPENUM_INPUT_{i}` : Typenum of the data in the array
- `ITEMSIZE_INPUT_{i}` : Size, in bytes, of the elements in the array.

In the same way, the macros `DTYPE_OUTPUT_{i}`, `ITEMSIZE_OUTPUT_{i}` and `TYPENUM_OUTPUT_{i}` are defined for every output 'i' of the op.

In addition to these macros, the `init_code_struct`, `code`, and `code_cleanup` section tags also have the following macros:

- `FAIL` : Code to insert at error points. A python exception should be set prior to this code. An invocation look like this:

  ```
  if (error) {
    // Set python exception
    FAIL
  }
  ```

  You can add a semicolon after the macro if it makes your editor happy.

- `CONTEXT` : Name of the context variable for this node. (only for Ops which have a context, which is discussed elsewhere)

Finally the tag `code` and `code_cleanup` have macros to pass the inputs and output names. These are name `INPUT_{i}` and `OUTPUT_{i}` where *i* is the 0-based index position in the input and output arrays respectively.

Support code Certain section are limited in what you can place in them due to semantic and syntactic restrictions of the C++ language. Most of these restrictions apply to the tags that end in `_struct`.

When we defined the VectorTimesVector op without using the `COp` class, we had to make a distinction between two types of support_code : the support code that was apply-specific and the support code that wasn't. The apply-specific code was defined in the `c_support_code_apply()` method and the elements defined in that code (global variables and functions) had to include the name of the Apply node in their own names to avoid conflicts between the different versions of the apply-specific code. The code that wasn't apply-specific was simply defined in the `c_support_code()` method.

To make indentifiers that include the *Apply* node name use the `APPLY_SPECIFIC(str)` macro. In the above example, this macro is used when defining the functions `vector_elemwise_mult()` and `vector_times_vector()` as well as when calling function `vector_elemwise_mult()` from inside `vector_times_vector()`.

When using the `COp` class, we still have to make the distinction between C code for each of the methods of a C class. These sections of code are separated by `#section <tag>` markers. The tag determines the name of the method this C code applies to with the rule that `<tag>` applies to *c_<tag>*. Unknown tags are an error and will be reported. Duplicate tags will be merged together in the order the appear in the C files.

The rules for knowing if where a piece of code should be put can be sometimes tricky. The key thing to remember is that things that can be shared between instances of the op should be apply-agnostic and go into a section which does not end in `_apply` or `_struct`. The distinction of `_apply` and `_struct` mostly hinghes on how you want to manange the lifetime of the object. Note that to use an apply-specific object, you have to be in a apply-specific section, so some portions of the code that might seem apply-agnostic may still be apply-specific because of the data they use (this does not include arguments).

In the above example, the `function vector_same_shape()` is apply-agnostic because it uses none of the macros defined by the class `COp` and it doesn't rely on any apply-specific code. The function `vector_elemwise_mult()` is apply-specific because it uses the macros defined by `COp`. Finally, the function `vector_times_vector()` is apply-specific because it uses those same macros and also because it calls `vector_elemwise_mult()` which is an apply-specific function.

Final Note

This tutorial focuses on providing C implementations to ops that manipulate Theano tensors. For more information about other Theano types, you can refer to the section *Alternate Theano Types*.

Python Memory Management

One of the major challenges in writing (somewhat) large-scale Python programs is to keep memory usage at a minimum. However, managing memory in Python is easy—if you just don't care. Python allocates memory transparently, manages objects using a reference count system, and frees memory when an object's reference count falls to zero. In theory, it's swell. In practice, you need to know a few things about Python memory management to get a memory-efficient program running. One of the things you should know, or at least get a good feel about, is the sizes of basic Python objects. Another thing is how Python manages its memory internally.

So let us begin with the size of basic objects. In Python, there's not a lot of primitive data types: there are ints, longs (an unlimited precision version of ints), floats (which are doubles), tuples, strings, lists, dictionaries, and classes.

Basic Objects

What is the size of `int`? A programmer with a C or C++ background will probably guess that the size of a machine-specific `int` is something like 32 bits, maybe 64; and that therefore it occupies at most 8 bytes. But is that so in Python?

Let us first write a function that shows the sizes of objects (recursively if necessary):

```
import sys

def show_sizeof(x, level=0):

    print "\t" * level, x.__class__, sys.getsizeof(x), x

    if hasattr(x, '__iter__'):
        if hasattr(x, 'items'):
            for xx in x.items():
                show_sizeof(xx, level + 1)
        else:
            for xx in x:
                show_sizeof(xx, level + 1)
```

We can now use the function to inspect the sizes of the different basic data types:

```
show_sizeof(None)
show_sizeof(3)
show_sizeof(2**63)
show_sizeof(102947298469128649161972364837164)
show_sizeof(9186593269437561348975613048756103487563847561934857613048756139485762974856984
```

If you have a 32-bit 2.7x Python, you'll see:

```
8 None
12 3
22 9223372036854775808
28 102947298469128649161972364837164
48 91865932694375613489756130487561034875638475619348576130487561394857629748569841
```

and if you have a 64-bit 2.7x Python, you'll see:

```
16 None
24 3
36 9223372036854775808
40 102947298469128649161972364837164
60 91865932694375613489756130487561034875638475619348576130487561394857629748569841
```

Let us focus on the 64-bit version (mainly because that's what we need the most often in our case). `None` takes 16 bytes. `int` takes 24 bytes, *three times* as much memory as a C `int64_t`, despite being some kind

of "machine-friendly" integer. Long integers (unbounded precision), used to represent integers larger than 2^{63}-1, have a minimum size of 36 bytes. Then it grows linearly in the logarithm of the integer represented.

Python's floats are implementation-specific but seem to be C doubles. However, they do not eat up only 8 bytes:

```
show_sizeof(3.14159265358979323846264338327950288)
```

Outputs

```
16 3.14159265359
```

on a 32-bit platform and

```
24 3.14159265359
```

on a 64-bit platform. That's again, three times the size a C programmer would expect. Now, what about strings?

```
show_sizeof("")
show_sizeof("My hovercraft is full of eels")
```

outputs, on a 32 bit platform:

```
21
50 My hovercraft is full of eels
```

and

```
37
66 My hovercraft is full of eels
```

An *empty* string costs 37 bytes in a 64-bit environment! Memory used by string then linearly grows in the length of the (useful) string.

* * *

Other structures commonly used, tuples, lists, and dictionaries are worthwhile to examine. Lists (which are implemented as array lists, not as linked lists, with everything it entails) are arrays of references to Python objects, allowing them to be heterogeneous. Let us look at our sizes:

```
show_sizeof([])
show_sizeof([4, "toaster", 230.1])
```

outputs

```
32 []
44 [4, 'toaster', 230.1]
```

on a 32-bit platform and

```
72 []
96 [4, 'toaster', 230.1]
```

on a 64-bit platform. An empty list eats up 72 bytes. The size of an empty, 64-bit C++ `std::list()` is only 16 bytes, 4-5 times less. What about tuples? (and dictionaries?):

```
show_sizeof({})
show_sizeof({'a':213, 'b':2131})
```

outputs, on a 32-bit box

```
136 {}
 136 {'a': 213, 'b': 2131}
        32 ('a', 213)
                22 a
                12 213
        32 ('b', 2131)
                22 b
                12 2131
```

and

```
280 {}
 280 {'a': 213, 'b': 2131}
        72 ('a', 213)
                38 a
                24 213
        72 ('b', 2131)
                38 b
                24 2131
```

for a 64-bit box.

This last example is particularly interesting because it "doesn't add up." If we look at individual key/value pairs, they take 72 bytes (while their components take 38+24=62 bytes, leaving 10 bytes for the pair itself), but the dictionary takes 280 bytes (rather than a strict minimum of 144=72×2 bytes). The dictionary is supposed to be an efficient data structure for search and the two likely implementations will use more space that strictly necessary. If it's some kind of tree, then we should pay the cost of internal nodes that contain a key and two pointers to children nodes; if it's a hash table, then we must have some room with free entries to ensure good performance.

The (somewhat) equivalent `std::map` C++ structure takes 48 bytes when created (that is, empty). An empty C++ string takes 8 bytes (then allocated size grows linearly the size of the string). An integer takes 4 bytes (32 bits).

* * *

Why does all this matter? It seems that whether an empty string takes 8 bytes or 37 doesn't change anything much. That's true. That's true *until* you need to scale. Then, you need to be really careful about how many objects you create to limit the quantity of memory your program uses. It is a problem in real-life applications. However, to devise a really good strategy about memory management, we must not only consider the sizes of objects, but how many and in which order they are created. It turns out to be very important for Python programs. One key element to understand is how Python allocates its memory internally, which we will discuss next.

Internal Memory Management

To speed-up memory allocation (and reuse) Python uses a number of lists for small objects. Each list will contain objects of similar size: there will be a list for objects 1 to 8 bytes in size, one for 9 to 16, etc. When a small object needs to be created, either we reuse a free block in the list, or we allocate a new one.

There are some internal details on how Python manages those lists into blocks, pools, and "arena": a number of block forms a pool, pools are gathered into arena, etc., but they're not very relevant to the point we want to make (if you really want to know, read Evan Jones' ideas on how to improve Python's memory allocation). The important point is that those lists *never shrink*.

Indeed: if an item (of size *x*) is deallocated (freed by lack of reference) its location is not returned to Python's global memory pool (and even less to the system), but merely marked as free and added to the free list of items of size *x*. The dead object's location will be reused if another object of compatible size is needed. If there are no dead objects available, new ones are created.

If small objects memory is never freed, then the inescapable conclusion is that, like goldfishes, these small object lists only keep growing, never shrinking, and that the memory footprint of your application is dominated by the largest number of small objects allocated at any given point.

* * *

Therefore, one should work hard to allocate only the number of small objects necessary for one task, favoring (otherwise *unpythonèsque*) loops where only a small number of elements are created/processed rather than (more *pythonèsque*) patterns where lists are created using list generation syntax then processed.

While the second pattern is more *à la Python*, it is rather the worst case: you end up creating lots of small objects that will come populate the small object lists, and even once the list is dead, the dead objects (now all in the free lists) will still occupy a lot of memory.

* * *

The fact that the free lists grow does not seem like much of a problem because the memory it contains is still accessible to the Python program. But from the OS's perspective, your program's size is the total (maximum) memory allocated to Python. Since Python returns memory to the OS on the heap (that allocates other objects than small objects) only on Windows, if you run on Linux, you can only see the total memory used by your program increase.

* * *

Let us prove my point using memory_profiler, a Python add-on module (which depends on the `python-psutil` package) by Fabian Pedregosa (the module's github page). This add-on provides the decorator `@profile` that allows one to monitor one specific function memory usage. It is extremely simple to use. Let us consider the following program:

```
import copy
import memory_profiler

@profile
def function():
    x = list(range(1000000))  # allocate a big list
    y = copy.deepcopy(x)
    del x
```

```
    return y

if __name__ == "__main__":
    function()
```

invoking

```
python -m memory_profiler memory-profile-me.py
```

prints, on a 64-bit computer

```
Filename: memory-profile-me.py

Line #    Mem usage    Increment   Line Contents
================================================
     4                             @profile
     5      9.11 MB      0.00 MB   def function():
     6     40.05 MB     30.94 MB       x = list(range(1000000)) # allocate a big list
     7     89.73 MB     49.68 MB       y = copy.deepcopy(x)
     8     82.10 MB     -7.63 MB       del x
     9     82.10 MB      0.00 MB       return y
```

This program creates a list of n=1,000,000 ints (n x 24 bytes = ~23 MB) and an additional list of references (n x 8 bytes = ~7.6 MB), which amounts to a total memory usage of ~31 MB. `copy.deepcopy` copies both lists, which allocates again ~50 MB (I am not sure where the additional overhead of 50 MB - 31 MB = 19 MB comes from). The interesting part is `del x`: it deletes `x`, but the memory usage only decreases by 7.63 MB! This is because `del` only deletes the reference list, not the actual integer values, which remain on the heap and cause a memory overhead of ~23 MB.

This example allocates in total ~73 MB, which is more than *twice* the amount of memory needed to store a single list of ~31 MB. You can see that memory can increase surprisingly if you are not careful!

Note that you might get different results on a different platform or with a different python version.

Pickle

On a related note: is `pickle` wasteful?

Pickle is the standard way of (de)serializing Python objects to file. What is its memory footprint? Does it create extra copies of the data or is it rather smart about it? Consider this short example:

```
import memory_profiler
import pickle
import random

def random_string():
    return "".join([chr(64 + random.randint(0, 25)) for _ in xrange(20)])

@profile
def create_file():
    x = [(random.random(),
          random_string(),
```

```
          random.randint(0, 2 ** 64))
         for _ in xrange(1000000)]

    pickle.dump(x, open('machin.pkl', 'w'))

@profile
def load_file():
    y = pickle.load(open('machin.pkl', 'r'))
    return y

if __name__=="__main__":
    create_file()
    #load_file()
```

With one invocation to profile the creation of the pickled data, and one invocation to re-read it (you comment out the function not to be called). Using `memory_profiler`, the creation uses a lot of memory:

```
Filename: test-pickle.py

Line #    Mem usage    Increment   Line Contents
================================================
     8                             @profile
     9      9.18 MB      0.00 MB   def create_file():
    10      9.33 MB      0.15 MB       x=[ (random.random(),
    11                                     random_string(),
    12                                     random.randint(0,2**64))
    13    246.11 MB    236.77 MB          for _ in xrange(1000000) ]
    14
    15    481.64 MB    235.54 MB       pickle.dump(x,open('machin.pkl','w'))
```

and re-reading a bit less:

```
Filename: test-pickle.py

Line #    Mem usage    Increment   Line Contents
================================================
    18                             @profile
    19      9.18 MB      0.00 MB   def load_file():
    20    311.02 MB    301.83 MB       y=pickle.load(open('machin.pkl','r'))
    21    311.02 MB      0.00 MB       return y
```

So somehow, *pickling* is very bad for memory consumption. The initial list takes up more or less 230MB, but pickling it creates an extra 230-something MB worth of memory allocation.

Unpickling, on the other hand, seems fairly efficient. It does create more memory than the original list (300MB instead of 230-something) but it does not double the quantity of allocated memory.

Overall, then, (un)pickling should be avoided for memory-sensitive applications. What are the alternatives? Pickling preserves all the structure of a data structure, so you can recover it exactly from the pickled file at a later time. However, that might not always be needed. If the file is to contain a list as in the example above, then maybe a simple flat, text-based, file format is in order. Let us see what it gives.

A naïve implementation would give:

```
import memory_profiler
import random
import pickle

def random_string():
    return "".join([chr(64 + random.randint(0, 25)) for _ in xrange(20)])

@profile
def create_file():
    x = [(random.random(),
          random_string(),
          random.randint(0, 2 ** 64))
         for _ in xrange(1000000) ]

    f = open('machin.flat', 'w')
    for xx in x:
        print >>f, xx
    f.close()

@profile
def load_file():
    y = []
    f = open('machin.flat', 'r')
    for line in f:
        y.append(eval(line))
    f.close()
    return y

if __name__== "__main__":
    create_file()
    #load_file()
```

Creating the file:

```
Filename: test-flat.py

Line #    Mem usage    Increment   Line Contents
================================================
     8                             @profile
     9      9.19 MB      0.00 MB   def create_file():
    10      9.34 MB      0.15 MB       x=[ (random.random(),
    11                                     random_string(),
    12                                     random.randint(0, 2**64))
    13    246.09 MB    236.75 MB          for _ in xrange(1000000) ]
    14
    15    246.09 MB      0.00 MB       f=open('machin.flat', 'w')
    16    308.27 MB     62.18 MB       for xx in x:
    17                                     print >>f, xx
```

and reading the file back:

```
Filename: test-flat.py

Line #    Mem usage    Increment   Line Contents
```

```
==============================================
    20                                 @profile
    21      9.19 MB      0.00 MB   def load_file():
    22      9.34 MB      0.15 MB       y=[]
    23      9.34 MB      0.00 MB       f=open('machin.flat', 'r')
    24    300.99 MB    291.66 MB       for line in f:
    25    300.99 MB      0.00 MB           y.append(eval(line))
    26    301.00 MB      0.00 MB       return y
```

Memory consumption on writing is now much better. It still creates a lot of temporary small objects (for 60MB's worth), but it's not doubling memory usage. Reading is comparable (using only marginally less memory).

This particular example is trivial but it generalizes to strategies where you don't load the whole thing first then process it but rather read a few items, process them, and reuse the allocated memory. Loading data to a Numpy array, for example, one could first create the Numpy array, then read the file line by line to fill the array: this allocates one copy of the whole data. Using pickle, you would allocate the whole data (at least) twice: once by pickle, and once through Numpy.

Or even better yet: use Numpy (or PyTables) arrays. But that's a different topic. In the mean time, you can have a look at loading and saving another tutorial in the Theano/doc/tutorial directory.

* * *

Python design goals are radically different than, say, C design goals. While the latter is designed to give you good control on what you're doing at the expense of more complex and explicit programming, the former is designed to let you code rapidly while hiding most (if not all) of the underlying implementation details. While this sounds nice, in a production environment ignoring the implementation inefficiencies of a language can bite you hard, and sometimes when it's too late. I think that having a good feel of how inefficient Python is with memory management (by design!) will play an important role in whether or not your code meets production requirements, scales well, or, on the contrary, will be a burning hell of memory.

Multi cores support in Theano

BLAS operation

BLAS is an interface for some mathematic operations between two vectors, a vector and a matrix or two matrices (e.g. the dot product between vector/matrix and matrix/matrix). Many different implementations of that interface exist and some of them are parallelized.

Theano tries to use that interface as frequently as possible for performance reasons. So if Theano links to a parallel implementation, those operations will run in parallel in Theano.

The most frequent way to control the number of threads used is via the `OMP_NUM_THREADS` environment variable. Set it to the number of threads you want to use before starting the Python process. Some BLAS implementations support other environment variables.

To test if you BLAS supports OpenMP/Multiple cores, you can use the theano/misc/check_blas.py script from the command line like this:

```
OMP_NUM_THREADS=1 python theano/misc/check_blas.py -q
OMP_NUM_THREADS=2 python theano/misc/check_blas.py -q
```

Parallel element wise ops with OpenMP

Because element wise ops work on every tensor entry independently they can be easily parallelized using OpenMP.

To use OpenMP you must set the `openmp` *flag* to `True`.

You can use the flag `openmp_elemwise_minsize` to set the minimum tensor size for which the operation is parallelized because for short tensors using OpenMP can slow down the operation. The default value is `200000`.

For simple (fast) operations you can obtain a speed-up with very large tensors while for more complex operations you can obtain a good speed-up also for smaller tensors.

There is a script `elemwise_openmp_speedup.py` in `theano/misc/` which you can use to tune the value of `openmp_elemwise_minsize` for your machine. The script runs two elemwise operations (a fast one and a slow one) for a vector of size `openmp_elemwise_minsize` with and without OpenMP and shows the time difference between the cases.

The only way to control the number of threads used is via the `OMP_NUM_THREADS` environment variable. Set it to the number of threads you want to use before starting the Python process. You can test this with this command:

```
OMP_NUM_THREADS=2 python theano/misc/elemwise_openmp_speedup.py
#The output

Fast op time without openmp 0.000533s with openmp 0.000474s speedup 1.12
Slow op time without openmp 0.002987s with openmp 0.001553s speedup 1.92
```

Frequently Asked Questions

How to update a subset of weights?

If you want to update only a subset of a weight matrix (such as some rows or some columns) that are used in the forward propogation of each iteration, then the cost function should be defined in a way that it only depends on the subset of weights that are used in that iteration.

For example if you want to learn a lookup table, e.g. used for word embeddings, where each row is a vector of weights representing the embedding that the model has learned for a word, in each iteration, the only rows that should get updated are those containing embeddings used during the forward propagation. Here is how the theano function should be written:

Defining a shared variable for the lookup table

```
>>> lookup_table = theano.shared(matrix_ndarray).
```

Getting a subset of the table (some rows or some columns) by passing an integer vector of indices corresponding to those rows or columns.

```
>>> subset = lookup_table[vector_of_indices]
```

From now on, use only 'subset'. Do not call lookup_table[vector_of_indices] again. This causes problems with grad as this will create new variables.

Defining cost which depends only on subset and not the entire lookup_table

```
>>> cost = something that depends on subset
>>> g = theano.grad(cost, subset)
```

There are two ways for updating the parameters: Either use inc_subtensor or set_subtensor. It is recommended to use inc_subtensor. Some theano optimizations do the conversion between the two functions, but not in all cases.

```
>>> updates = inc_subtensor(subset, g*lr)
OR
>>> updates = set_subtensor(subset, subset + g*lr)
```

Currently we just cover the case here, not if you use inc_subtensor or set_subtensor with other types of indexing.

Defining the theano function

```
>>> f=theano.function(..., updates=updates)
```

Note that you can compute the gradient of the cost function w.r.t. the entire lookup_table, and the gradient will have nonzero rows only for the rows that were selected during forward propagation. If you use gradient descent to update the parameters, there are no issues except for unnecessary computation, e.g. you will update the lookup table parameters with many zero gradient rows. However, if you want to use a different optimization method like rmsprop or Hessian-Free optimization, then there will be issues. In rmsprop, you keep an exponentially decaying squared gradient by whose square root you divide the current gradient to rescale the update step component-wise. If the gradient of the lookup table row which corresponds to a rare word is very often zero, the squared gradient history will tend to zero for that row because the history of that row decays towards zero. Using Hessian-Free, you will get many zero rows and columns. Even one of them would make it non-invertible. In general, it would be better to compute the gradient only w.r.t. to those lookup table rows or columns which are actually used during the forward propagation.

6.2.5 Library Documentation

This documentation covers Theano module-wise. This is suited to finding the Types and Ops that you can use to build and compile expression graphs.

`tensor` – Types and Ops for Symbolic numpy

Theano's strength is in expressing symbolic calculations involving tensors. There are many types of symbolic expressions for tensors. They are grouped into the following sections:

Basic Tensor Functionality

Theano supports any kind of Python object, but its focus is support for symbolic matrix expressions. When you type,

```
>>> x = T.fmatrix()
```

the `x` is a `TensorVariable` instance. The `T.fmatrix` object itself is an instance of `TensorType`. Theano knows what type of variable `x` is because `x.type` points back to `T.fmatrix`.

This chapter explains the various ways of creating tensor variables, the attributes and methods of `TensorVariable` and `TensorType`, and various basic symbolic math and arithmetic that Theano supports for tensor variables.

Creation Theano provides a list of predefined tensor types that can be used to create a tensor variables. Variables can be named to facilitate debugging, and all of these constructors accept an optional `name` argument. For example, the following each produce a TensorVariable instance that stands for a 0-dimensional ndarray of integers with the name `'myvar'`:

```
>>> x = scalar('myvar', dtype='int32')
>>> x = iscalar('myvar')
>>> x = TensorType(dtype='int32', broadcastable=())('myvar')
```

Constructors with optional dtype These are the simplest and often-preferred methods for creating symbolic variables in your code. By default, they produce floating-point variables (with dtype determined by config.floatX, see `floatX`) so if you use these constructors it is easy to switch your code between different levels of floating-point precision.

`theano.tensor.`**`scalar`**(*name=None*, *dtype=config.floatX*)
Return a Variable for a 0-dimensional ndarray

`theano.tensor.`**`vector`**(*name=None*, *dtype=config.floatX*)
Return a Variable for a 1-dimensional ndarray

`theano.tensor.`**`row`**(*name=None*, *dtype=config.floatX*)
Return a Variable for a 2-dimensional ndarray in which the number of rows is guaranteed to be 1.

`theano.tensor.`**`col`**(*name=None*, *dtype=config.floatX*)
Return a Variable for a 2-dimensional ndarray in which the number of columns is guaranteed to be 1.

`theano.tensor.`**`matrix`**(*name=None*, *dtype=config.floatX*)
Return a Variable for a 2-dimensional ndarray

`theano.tensor.`**`tensor3`**(*name=None*, *dtype=config.floatX*)
Return a Variable for a 3-dimensional ndarray

`theano.tensor.`**`tensor4`**(*name=None*, *dtype=config.floatX*)
Return a Variable for a 4-dimensional ndarray

All Fully-Typed Constructors The following TensorType instances are provided in the theano.tensor module. They are all callable, and accept an optional `name` argument. So for example:

```
from theano.tensor import *

x = dmatrix()         # creates one Variable with no name
x = dmatrix('x')      # creates one Variable with name 'x'
xyz = dmatrix('xyz') # creates one Variable with name 'xyz'
```

Constructor	dtype	ndim	shape	broadcastable
bscalar	int8	0	()	()
bvector	int8	1	(?,)	(False,)
brow	int8	2	(1,?)	(True, False)
bcol	int8	2	(?,1)	(False, True)
bmatrix	int8	2	(?,?)	(False, False)
btensor3	int8	3	(?,?,?)	(False, False, False)
btensor4	int8	4	(?,?,?,?)	(False, False, False, False)
wscalar	int16	0	()	()
wvector	int16	1	(?,)	(False,)
wrow	int16	2	(1,?)	(True, False)
wcol	int16	2	(?,1)	(False, True)
wmatrix	int16	2	(?,?)	(False, False)
wtensor3	int16	3	(?,?,?)	(False, False, False)
wtensor4	int16	4	(?,?,?,?)	(False, False, False, False)
iscalar	int32	0	()	()
ivector	int32	1	(?,)	(False,)
irow	int32	2	(1,?)	(True, False)
icol	int32	2	(?,1)	(False, True)
imatrix	int32	2	(?,?)	(False, False)
itensor3	int32	3	(?,?,?)	(False, False, False)
itensor4	int32	4	(?,?,?,?)	(False, False, False, False)
lscalar	int64	0	()	()
lvector	int64	1	(?,)	(False,)
lrow	int64	2	(1,?)	(True, False)
lcol	int64	2	(?,1)	(False, True)
lmatrix	int64	2	(?,?)	(False, False)
ltensor3	int64	3	(?,?,?)	(False, False, False)
ltensor4	int64	4	(?,?,?,?)	(False, False, False, False)
dscalar	float64	0	()	()
dvector	float64	1	(?,)	(False,)
drow	float64	2	(1,?)	(True, False)
dcol	float64	2	(?,1)	(False, True)
dmatrix	float64	2	(?,?)	(False, False)
dtensor3	float64	3	(?,?,?)	(False, False, False)
dtensor4	float64	4	(?,?,?,?)	(False, False, False, False)
fscalar	float32	0	()	()

Continued on next page

Table 6.1 – continued from previous page

Constructor	dtype	ndim	shape	broadcastable
fvector	float32	1	(?,)	(False,)
frow	float32	2	(1,?)	(True, False)
fcol	float32	2	(?,1)	(False, True)
fmatrix	float32	2	(?,?)	(False, False)
ftensor3	float32	3	(?,?,?)	(False, False, False)
ftensor4	float32	4	(?,?,?,?)	(False, False, False, False)
cscalar	complex64	0	()	()
cvector	complex64	1	(?,)	(False,)
crow	complex64	2	(1,?)	(True, False)
ccol	complex64	2	(?,1)	(False, True)
cmatrix	complex64	2	(?,?)	(False, False)
ctensor3	complex64	3	(?,?,?)	(False, False, False)
ctensor4	complex64	4	(?,?,?,?)	(False, False, False, False)
zscalar	complex128	0	()	()
zvector	complex128	1	(?,)	(False,)
zrow	complex128	2	(1,?)	(True, False)
zcol	complex128	2	(?,1)	(False, True)
zmatrix	complex128	2	(?,?)	(False, False)
ztensor3	complex128	3	(?,?,?)	(False, False, False)
ztensor4	complex128	4	(?,?,?,?)	(False, False, False, False)

Plural Constructors There are several constructors that can produce multiple variables at once. These are not frequently used in practice, but often used in tutorial examples to save space!

iscalars, lscalars, fscalars, dscalars
Return one or more scalar variables.

ivectors, lvectors, fvectors, dvectors
Return one or more vector variables.

irows, lrows, frows, drows
Return one or more row variables.

icols, lcols, fcols, dcols
Return one or more col variables.

imatrices, lmatrices, fmatrices, dmatrices
Return one or more matrix variables.

Each of these plural constructors accepts an integer or several strings. If an integer is provided, the method will return that many Variables and if strings are provided, it will create one Variable for each string, using the string as the Variable's name. For example:

```
from theano.tensor import *

x, y, z = dmatrices(3) # creates three matrix Variables with no names
x, y, z = dmatrices('x', 'y', 'z') # creates three matrix Variables named 'x', 'y' and 'z'
```

Custom tensor types If you would like to construct a tensor variable with a non-standard broadcasting pattern, or a larger number of dimensions you'll need to create your own `TensorType` instance. You create such an instance by passing the dtype and broadcasting pattern to the constructor. For example, you can create your own 5-dimensional tensor type

```
>>> dtensor5 = TensorType('float64', (False,)*5)
>>> x = dtensor5()
>>> z = dtensor5('z')
```

You can also redefine some of the provided types and they will interact correctly:

```
>>> my_dmatrix = TensorType('float64', (False,)*2)
>>> x = my_dmatrix()        # allocate a matrix variable
>>> my_dmatrix == dmatrix
True
```

See `TensorType` for more information about creating new types of Tensor.

Converting from Python Objects Another way of creating a TensorVariable (a TensorSharedVariable to be precise) is by calling `shared()`

```
x = shared(numpy.random.randn(3,4))
```

This will return a *shared variable* whose `.value` is a numpy ndarray. The number of dimensions and dtype of the Variable are inferred from the ndarray argument. The argument to *shared will not be copied*, and subsequent changes will be reflected in `x.value`.

For additional information, see the `shared()` documentation. Finally, when you use a numpy ndarry or a Python number together with `TensorVariable` instances in arithmetic expressions, the result is a `TensorVariable`. What happens to the ndarray or the number? Theano requires that the inputs to all expressions be Variable instances, so Theano automatically wraps them in a `TensorConstant`.

Note: Theano makes a copy of any ndarray that you use in an expression, so subsequent changes to that ndarray will not have any effect on the Theano expression.

For numpy ndarrays the dtype is given, but the broadcastable pattern must be inferred. The TensorConstant is given a type with a matching dtype, and a broadcastable pattern with a `True` for every shape dimension that is 1.

For python numbers, the broadcastable pattern is `()` but the dtype must be inferred. Python integers are stored in the smallest dtype that can hold them, so small constants like `1` are stored in a `bscalar`. Likewise, Python floats are stored in an fscalar if fscalar suffices to hold them perfectly, but a dscalar otherwise.

Note: When config.floatX==float32 (see `config`), then Python floats are stored instead as single-precision floats.

For fine control of this rounding policy, see theano.tensor.basic.autocast_float.

`theano.tensor.`**`as_tensor_variable`**(*x*, *name=None*, *ndim=None*)

Turn an argument *x* into a TensorVariable or TensorConstant.

Many tensor Ops run their arguments through this function as pre-processing. It passes through TensorVariable instances, and tries to wrap other objects into TensorConstant.

When *x* is a Python number, the dtype is inferred as described above.

When *x* is a *list* or *tuple* it is passed through numpy.asarray

If the *ndim* argument is not None, it must be an integer and the output will be broadcasted if necessary in order to have this many dimensions.

Return type `TensorVariable` or `TensorConstant`

TensorType and TensorVariable

class `theano.tensor.`**`TensorType`**(*Type*)

The Type class used to mark Variables that stand for *numpy.ndarray* values (*numpy.memmap*, which is a subclass of *numpy.ndarray*, is also allowed). Recalling to the tutorial, the purple box in *the tutorial's graph-structure figure* is an instance of this class.

`broadcastable`

A tuple of True/False values, one for each dimension. True in position 'i' indicates that at evaluation-time, the ndarray will have size 1 in that 'i'-th dimension. Such a dimension is called a *broadcastable dimension* (see *Broadcasting in Theano vs. Numpy*).

The broadcastable pattern indicates both the number of dimensions and whether a particular dimension must have length 1.

Here is a table mapping some *broadcastable* patterns to what they mean:

pattern	interpretation
[]	scalar
[True]	1D scalar (vector of length 1)
[True, True]	2D scalar (1x1 matrix)
[False]	vector
[False, False]	matrix
[False] * n	nD tensor
[True, False]	row (1xN matrix)
[False, True]	column (Mx1 matrix)
[False, True, False]	A Mx1xP tensor (a)
[True, False, False]	A 1xNxP tensor (b)
[False, False, False]	A MxNxP tensor (pattern of a + b)

For dimensions in which broadcasting is False, the length of this dimension can be 1 or more. For dimensions in which broadcasting is True, the length of this dimension must be 1.

When two arguments to an element-wise operation (like addition or subtraction) have a different number of dimensions, the broadcastable pattern is *expanded to the left*, by padding with `True`. For example, a vector's pattern, `[False]`, could be expanded to `[True, False]`, and would behave like a row (1xN matrix). In the same way, a matrix (`[False, False]`) would behave like a 1xNxP tensor (`[True, False, False]`).

If we wanted to create a type representing a matrix that would broadcast over the middle dimension of a 3-dimensional tensor when adding them together, we would define it like this:

```
>>> middle_broadcaster = TensorType('complex64', [False, True, False])
```

ndim
The number of dimensions that a Variable's value will have at evaluation-time. This must be known when we are building the expression graph.

dtype
A string indicating the numerical type of the ndarray for which a Variable of this Type is standing. The dtype attribute of a TensorType instance can be any of the following strings.

dtype	domain	bits
'int8'	signed integer	8
'int16'	signed integer	16
'int32'	signed integer	32
'int64'	signed integer	64
'uint8'	unsigned integer	8
'uint16'	unsigned integer	16
'uint32'	unsigned integer	32
'uint64'	unsigned integer	64
'float32'	floating point	32
'float64'	floating point	64
'complex64'	complex	64 (two float32)
'complex128'	complex	128 (two float64)

__init__(*self*, *dtype*, *broadcastable*)
If you wish to use a type of tensor which is not already available (for example, a 5D tensor) you can build an appropriate type by instantiating `TensorType`.

TensorVariable

class `theano.tensor.`**TensorVariable**(*Variable*, *_tensor_py_operators*)
The result of symbolic operations typically have this type.

See `_tensor_py_operators` for most of the attributes and methods you'll want to call.

class `theano.tensor.`**TensorConstant**(*Variable*, *_tensor_py_operators*)
Python and numpy numbers are wrapped in this type.

See `_tensor_py_operators` for most of the attributes and methods you'll want to call.

class `theano.tensor.`**TensorSharedVariable**(*Variable*, *_tensor_py_operators*)
This type is returned by `shared()` when the value to share is a numpy ndarray.

See `_tensor_py_operators` for most of the attributes and methods you'll want to call.

class `theano.tensor.`**_tensor_py_operators**(*object*)
This mix-in class adds convenient attributes, methods, and support to TensorVariable, TensorConstant and TensorSharedVariable for Python operators (see *Operator Support*).

type
A reference to the `TensorType` instance describing the sort of values that might be associated with this variable.

ndim
The number of dimensions of this tensor. Aliased to `TensorType.ndim`.

dtype
The numeric type of this tensor. Aliased to `TensorType.dtype`.

reshape(*shape*, *ndim=None*)
Returns a view of this tensor that has been reshaped as in numpy.reshape. If the shape is a Variable argument, then you might need to use the optional *ndim* parameter to declare how many elements the shape has, and therefore how many dimensions the reshaped Variable will have.

See `reshape()`.

dimshuffle(**pattern*)
Returns a view of this tensor with permuted dimensions. Typically the pattern will include the integers 0, 1, ... ndim-1, and any number of 'x' characters in dimensions where this tensor should be broadcasted.

A few examples of patterns and their effect:

- ('x') -> make a 0d (scalar) into a 1d vector
- (0, 1) -> identity for 2d vectors
- (1, 0) -> inverts the first and second dimensions
- ('x', 0) -> make a row out of a 1d vector (N to 1xN)
- (0, 'x') -> make a column out of a 1d vector (N to Nx1)
- (2, 0, 1) -> AxBxC to CxAxB
- (0, 'x', 1) -> AxB to Ax1xB
- (1, 'x', 0) -> AxB to Bx1xA
- (1,) -> This remove dimensions 0. It must be a broadcastable dimension (1xA to A)

flatten(*ndim=1*)
Returns a view of this tensor with *ndim* dimensions, whose shape for the first *ndim-1* dimensions will be the same as *self*, and shape in the remaining dimension will be expanded to fit in all the data from self.

See `flatten()`.

ravel()
return self.flatten(). For NumPy compatibility.

T
Transpose of this tensor.

```
>>> x = T.zmatrix()
>>> y = 3+.2j * x.T
```

Note: In numpy and in Theano, the transpose of a vector is exactly the same vector! Use *reshape* or *dimshuffle* to turn your vector into a row or column matrix.

{any,all}(axis=None, keepdims=False)

{sum,prod,mean}(axis=None, dtype=None, keepdims=False, acc_dtype=None)

{var,std,min,max,argmin,argmax}(axis=None, keepdims=False),

diagonal(*offset=0*, *axis1=0*, *axis2=1*)

astype(*dtype*)

take(*indices*, *axis=None*, *mode='raise'*)

copy()

norm(*L*, *axis=None*)

nonzero(*self*, *return_matrix=False*)

nonzero_values(*self*)

sort(*self*, *axis=-1*, *kind='quicksort'*, *order=None*)

argsort(*self*, *axis=-1*, *kind='quicksort'*, *order=None*)

clip(*self*, *a_min*, *a_max*)

conf()

repeat(*repeats*, *axis=None*)

round(*mode="half_away_from_zero"*)

trace()

get_scalar_constant_value()

zeros_like(*model*, *dtype=None*)
All the above methods are equivalent to NumPy for Theano on the current tensor.

__{abs,neg,lt,le,gt,ge,invert,and,or,add,sub,mul,div,truediv,floordiv}__
Those elemwise operation are supported via Python syntax.

Shaping and Shuffling To re-order the dimensions of a variable, to insert or remove broadcastable dimensions, see `_tensor_py_operators.dimshuffle()`.

theano.tensor.**shape**(*x*)
Returns an lvector representing the shape of *x*.

theano.tensor.**reshape**(*x*, *newshape*, *ndim=None*)

Parameters

- **x** (*any TensorVariable (or compatible)*) – variable to be reshaped
- **newshape** (*lvector (or compatible)*) – the new shape for *x*
- **ndim** – optional - the length that *newshape*'s value will have. If this is `None`, then *reshape()* will infer it from *newshape*.

Return type variable with x's dtype, but ndim dimensions

Note: This function can infer the length of a symbolic newshape in some cases, but if it cannot and you do not provide the *ndim*, then this function will raise an Exception.

---

theano.tensor.**shape_padleft**(*x*, *n_ones=1*)

Reshape *x* by left padding the shape with *n_ones* 1s. Note that all this new dimension will be broadcastable. To make them non-broadcastable see the `unbroadcast()`.

Parameters **x** (*any TensorVariable (or compatible)*) – variable to be reshaped

theano.tensor.**shape_padright**(*x*, *n_ones=1*)

Reshape *x* by right padding the shape with *n_ones* 1s. Note that all this new dimension will be broadcastable. To make them non-broadcastable see the `unbroadcast()`.

Parameters **x** (*any TensorVariable (or compatible)*) – variable to be reshaped

theano.tensor.**unbroadcast**(*x*, **axes*)

Make the input impossible to broadcast in the specified axes. For example, addbroadcast(x, 0) will make the first dimension of x broadcastable. When performing the function, if the length of x along that dimension is not 1, a ValueError will be raised.

We apply the opt here not to pollute the graph especially during the gpu optimization

x [tensor_like] Input theano tensor.

axis [an int or an iterable object such as list or tuple] of int values

The dimension along which the tensor x should be unbroadcastable. if the length of x along these dimensions is not 1, a ValueError will be raised.

a theano tensor, which is unbroadcastable along the specified dimensions.

theano.tensor.**addbroadcast**(*x*, **axes*)

Make the input broadcastable in the specified axes. For example, addbroadcast(x, 0) will make the first dimension of x broadcastable. When performing the function, if the length of x along that dimension is not 1, a ValueError will be raised.

We apply the opt here not to pollute the graph especially during the gpu optimization

x [tensor_like] Input theano tensor.

axis [an int or an iterable object such as list or tuple] of int values

The dimension along which the tensor x should be broadcastable. if the length of x along these dimensions is not 1, a ValueError will be raised.

a theano tensor, which is broadcastable along the specified dimensions.

theano.tensor.**patternbroadcast**(*x*, *broadcastable*)

Make the input adopt a specific broadcasting pattern. broadcastable must be iterable. For example, patternbroadcast(x, (True, False)) will make the first dimension of x broadcastable and the second dimension not broadcastable, so x will now be a row.

We apply the opt here not to pollute the graph especially during the gpu optimization.

x [tensor_like] Input theano tensor.

broadcastable [an iterable object such as list or tuple]

of bool values

a set of boolean values indicating whether a dimension should be broadcastable or not. if the length of x along these dimensions is not 1, a ValueError will be raised.

a theano tensor, which is unbroadcastable along the specified dimensions.

`theano.tensor.`**`flatten`**(*x*, *outdim=1*)

Similar to `reshape()`, but the shape is inferred from the shape of *x*.

Parameters

- **x** (*any TensorVariable (or compatible)*) – variable to be flattened
- **outdim** (*int*) – the number of dimensions in the returned variable

Return type variable with same dtype as *x* and *outdim* dimensions

Returns variable with the same shape as *x* in the leading *outdim-1* dimensions, but with all remaining dimensions of *x* collapsed into the last dimension.

For example, if we flatten a tensor of shape (2, 3, 4, 5) with flatten(x, outdim=2), then we'll have the same (2-1=1) leading dimensions (2,), and the remaining dimensions are collapsed. So the output in this example would have shape (2, 60).

`theano.tensor.`**`tile`**(*x*, *reps*, *ndim=None*)

Construct an array by repeating the input *x* according to *reps* pattern.

Tiles its input according to *reps*. The length of *reps* is the number of dimension of *x* and contains the number of times to tile *x* in each dimension.

See numpy.tile documentation for examples.

See `theano.tensor.extra_ops.repeat`

Note Currently, *reps* must be a constant, *x.ndim* and *len(reps)* must be equal and, if specified, *ndim* must be equal to both.

Creating Tensor

`theano.tensor.`**`zeros_like`**(*x*)

Parameters **x** – tensor that has same shape as output

Returns a tensor filled with 0s that has same shape as *x*.

`theano.tensor.`**`ones_like`**(*x*)

Parameters **x** – tensor that has same shape as output

Returns a tensor filled with 1s that has same shape as *x*.

`theano.tensor.`**`fill`**(*a*, *b*)

Parameters

- **a** – tensor that has same shape as output
- **b** – theano scalar or value with which you want to fill the output

Create a matrix by filling the shape of *a* with *b*

theano.tensor.**alloc**(*value*, **shape*)

Parameters

- **value** – a value with which to fill the output
- **shape** – the dimensions of the returned array

Returns an N-dimensional tensor initialized by *value* and having the specified shape.

theano.tensor.**eye**(*n*, *m=None*, *k=0*, *dtype=theano.config.floatX*)

Parameters

- **n** – number of rows in output (value or theano scalar)
- **m** – number of columns in output (value or theano scalar)
- **k** – Index of the diagonal: 0 refers to the main diagonal, a positive value refers to an upper diagonal, and a negative value to a lower diagonal. It can be a theano scalar.

Returns An array where all elements are equal to zero, except for the *k*-th diagonal, whose values are equal to one.

theano.tensor.**identity_like**(*x*)

Parameters **x** – tensor

Returns A tensor of same shape as *x* that is filled with 0s everywhere except for the main diagonal, whose values are equal to one. The output will have same dtype as *x*.

theano.tensor.**stack**(**tensors*)

Return a Tensor representing for the arguments all stacked up into a single Tensor. (of 1 rank greater).

Parameters **tensors** – one or more tensors of the same rank

Returns A tensor such that rval[0] == tensors[0], rval[1] == tensors[1], etc.

```
>>> x0 = T.scalar()
>>> x1 = T.scalar()
>>> x2 = T.scalar()
>>> x = T.stack(x0, x1, x2)
>>> x.ndim # x is a vector of length 3.
1
```

theano.tensor.**concatenate**(*tensor_list*, *axis=0*)

Parameters

- **tensor_list** (a list or tuple of Tensors that all have the same shape in the axes *not* specified by the *axis* argument.) – one or more Tensors to be concatenated together into one.
- **axis** (*literal or symbolic integer*) – Tensors will be joined along this axis, so they may have different `shape[axis]`

```
>>> x0 = T.fmatrix()
>>> x1 = T.ftensor3()
>>> x2 = T.fvector()
>>> x = T.concatenate([x0, x1[0], T.shape_padright(x2)], axis=1)
>>> x.ndim
2
```

`theano.tensor.`**`stacklists`**(*tensor_list*)

Parameters **tensor_list** (an iterable that contains either tensors or other iterables of the same type as *tensor_list* (in other words, this is a tree whose leaves are tensors).) – tensors to be stacked together.

Recursively stack lists of tensors to maintain similar structure.

This function can create a tensor from a shaped list of scalars:

```
>>> from theano.tensor import stacklists, scalars, matrices
>>> from theano import function
>>> a, b, c, d = scalars('abcd')
>>> X = stacklists([[a, b], [c, d]])
>>> f = function([a, b, c, d], X)
>>> f(1, 2, 3, 4)
array([[ 1.,  2.],
       [ 3.,  4.]])
```

We can also stack arbitrarily shaped tensors. Here we stack matrices into a 2 by 2 grid:

```
>>> from numpy import ones
>>> a, b, c, d = matrices('abcd')
>>> X = stacklists([[a, b], [c, d]])
>>> f = function([a, b, c, d], X)
>>> x = ones((4, 4), 'float32')
>>> f(x, x, x, x).shape
(2, 2, 4, 4)
```

`theano.tensor.basic.`**`choose`**(*a*, *choices*, *out=None*, *mode='raise'*)

Construct an array from an index array and a set of arrays to choose from.

First of all, if confused or uncertain, definitely look at the Examples - in its full generality, this function is less simple than it might seem from the following code description (below ndi = numpy.lib.index_tricks):

np.choose(a,c) == np.array([c[a[I]][I] for I in ndi.ndindex(a.shape)]).

But this omits some subtleties. Here is a fully general summary:

Given an `index` array (a) of integers and a sequence of n arrays (choices), a and each choice array are first broadcast, as necessary, to arrays of a common shape; calling these Ba and Bchoices[i], i =

0,...,n-1 we have that, necessarily, Ba.shape == Bchoices[i].shape for each i. Then, a new array with shape Ba.shape is created as follows:

- if mode=raise (the default), then, first of all, each element of a (and thus Ba) must be in the range [0, n-1]; now, suppose that i (in that range) is the value at the (j0, j1, ..., jm) position in Ba - then the value at the same position in the new array is the value in Bchoices[i] at that same position;
- if mode=wrap, values in a (and thus Ba) may be any (signed) integer; modular arithmetic is used to map integers outside the range [0, n-1] back into that range; and then the new array is constructed as above;
- if mode=clip, values in a (and thus Ba) may be any (signed) integer; negative integers are mapped to 0; values greater than n-1 are mapped to n-1; and then the new array is constructed as above.

Parameter *a* - int array This array must contain integers in [0, n-1], where n is the number of choices, unless mode=wrap or mode=clip, in which cases any integers are permissible.

Parameter *choices* - sequence of arrays Choice arrays. a and all of the choices must be broadcastable to the same shape. If choices is itself an array (not recommended), then its outermost dimension (i.e., the one corresponding to choices.shape[0]) is taken as defining the `sequence`.

Parameter *out* - array, optional If provided, the result will be inserted into this array. It should be of the appropriate shape and dtype.

Parameter *mode* - {`raise` (default), `wrap`, `clip`}, optional Specifies how indices outside [0, n-1] will be treated: `raise` : an exception is raised `wrap` : value becomes value mod n `clip` : values < 0 are mapped to 0, values > n-1 are mapped to n-1

Returns merged_array - array The merged result.

Raises ValueError - shape mismatch If a and each choice array are not all broadcastable to the same shape.

Reductions

`theano.tensor.`**`max`**(*x*, *axis=None*, *keepdims=False*)

Parameter *x* - symbolic Tensor (or compatible)

Parameter *axis* - axis or axes along which to compute the maximum

Parameter *keepdims* - (boolean) If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original tensor.

Returns maximum of *x* along *axis*

axis can be:

- *None* - in which case the maximum is computed along all axes (like numpy)
- an *int* - computed along this axis

- a *list of ints* - computed along these axes

theano.tensor.**argmax**(*x*, *axis=None*, *keepdims=False*)

Parameter *x* - symbolic Tensor (or compatible)

Parameter *axis* - axis along which to compute the index of the maximum

Parameter *keepdims* - (boolean) If this is set to True, the axis which is reduced is left in the result as a dimension with size one. With this option, the result will broadcast correctly against the original tensor.

Returns the index of the maximum value along a given axis

if axis=None, Theano 0.5rc1 or later: argmax over the flattened tensor (like numpy) older: then axis is assumed to be ndim(x)-1

theano.tensor.**max_and_argmax**(*x*, *axis=None*, *keepdims=False*)

Parameter *x* - symbolic Tensor (or compatible)

Parameter *axis* - axis along which to compute the maximum and its index

Parameter *keepdims* - (boolean) If this is set to True, the axis which is reduced is left in the result as a dimension with size one. With this option, the result will broadcast correctly against the original tensor.

Returns the maxium value along a given axis and its index.

if axis=None, Theano 0.5rc1 or later: max_and_argmax over the flattened tensor (like numpy) older: then axis is assumed to be ndim(x)-1

theano.tensor.**min**(*x*, *axis=None*, *keepdims=False*)

Parameter *x* - symbolic Tensor (or compatible)

Parameter *axis* - axis or axes along which to compute the minimum

Parameter *keepdims* - (boolean) If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original tensor.

Returns minimum of *x* along *axis*

axis can be:

- *None* - in which case the minimum is computed along all axes (like numpy)
- an *int* - computed along this axis
- a *list of ints* - computed along these axes

theano.tensor.**argmin**(*x*, *axis=None*, *keepdims=False*)

Parameter *x* - symbolic Tensor (or compatible)

Parameter *axis* - axis along which to compute the index of the minimum

Parameter *keepdims* - (boolean) If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original tensor.

Returns the index of the minimum value along a given axis

if axis=None, Theano 0.5rc1 or later: argmin over the flattened tensor (like numpy) older: then axis is assumed to be ndim(x)-1

theano.tensor.**sum**(*x*, *axis=None*, *dtype=None*, *keepdims=False*, *acc_dtype=None*)

Parameter *x* - symbolic Tensor (or compatible)

Parameter *axis* - axis or axes along which to compute the sum

Parameter *dtype* - The dtype of the returned tensor. If None, then we use the default dtype which is the same as the input tensor's dtype except when:

- the input dtype is a signed integer of precision < 64 bit, in which case we use int64
- the input dtype is an unsigned integer of precision < 64 bit, in which case we use uint64

This default dtype does _not_ depend on the value of "acc_dtype".

Parameter *keepdims* - (boolean) If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original tensor.

Parameter *acc_dtype* - The dtype of the internal accumulator. If None (default), we use the dtype in the list below, or the input dtype if its precision is higher:

- for int dtypes, we use at least int64;
- for uint dtypes, we use at least uint64;
- for float dtypes, we use at least float64;
- for complex dtypes, we use at least complex128.

Returns sum of *x* along *axis*

axis can be:

- *None* - in which case the sum is computed along all axes (like numpy)
- an *int* - computed along this axis
- a *list of ints* - computed along these axes

theano.tensor.**prod**(*x*, *axis=None*, *dtype=None*, *keepdims=False*, *acc_dtype=None*, *no_zeros_in_input=False*)

Parameter *x* - symbolic Tensor (or compatible)

Parameter *axis* - axis or axes along which to compute the product

Parameter *dtype* - The dtype of the returned tensor. If None, then we use the default dtype which is the same as the input tensor's dtype except when:

- the input dtype is a signed integer of precision < 64 bit, in which case we use int64
- the input dtype is an unsigned integer of precision < 64 bit, in which case we use uint64

This default dtype does _not_ depend on the value of "acc_dtype".

Parameter *keepdims* - (boolean) If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original tensor.

Parameter *acc_dtype* - The dtype of the internal accumulator. If None (default), we use the dtype in the list below, or the input dtype if its precision is higher:

- for int dtypes, we use at least int64;
- for uint dtypes, we use at least uint64;
- for float dtypes, we use at least float64;
- for complex dtypes, we use at least complex128.

Parameter *no_zeros_in_input* - The grad of prod is complicated as we need to handle 3 different cases: without zeros in the input reduced group, with 1 zero or with more zeros.

This could slow you down, but more importantly, we currently don't support the second derivative of the 3 cases. So you cannot take the second derivative of the default prod().

To remove the handling of the special cases of 0 and so get some small speed up and allow second derivative set `no_zeros_in_inputs` to `True`. It defaults to `False`.

It is the user responsibility to make sure there are no zeros in the inputs. If there are, the grad will be wrong.

Returns product of every term in *x* along *axis*

axis can be:

- *None* - in which case the sum is computed along all axes (like numpy)
- an *int* - computed along this axis
- a *list of ints* - computed along these axes

`theano.tensor.`**`mean`**(*x*, *axis=None*, *dtype=None*, *keepdims=False*, *acc_dtype=None*)

Parameter *x* - symbolic Tensor (or compatible)

Parameter *axis* - axis or axes along which to compute the mean

Parameter *dtype* - The dtype to cast the result of the inner summation into. For instance, by default, a sum of a float32 tensor will be done in float64 (acc_dtype would be float64 by default), but that result will be casted back in float32.

Parameter *keepdims* - (boolean) If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original tensor.

Parameter *acc_dtype* - The dtype of the internal accumulator of the inner summation. This will not necessarily be the dtype of the output (in particular if it is a discrete (int/uint) dtype, the output will be in a float type). If None, then we use the same rules as `sum()`.

Returns mean value of *x* along *axis*

axis can be:

- *None* - in which case the mean is computed along all axes (like numpy)
- an *int* - computed along this axis
- a *list of ints* - computed along these axes

`theano.tensor.`**`var`**(*x*, *axis=None*, *keepdims=False*)

Parameter *x* - symbolic Tensor (or compatible)

Parameter *axis* - axis or axes along which to compute the variance

Parameter *keepdims* - (boolean) If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original tensor.

Returns variance of *x* along *axis*

axis can be:

- *None* - in which case the variance is computed along all axes (like numpy)
- an *int* - computed along this axis
- a *list of ints* - computed along these axes

`theano.tensor.`**`std`**(*x*, *axis=None*, *keepdims=False*)

Parameter *x* - symbolic Tensor (or compatible)

Parameter *axis* - axis or axes along which to compute the standard deviation

Parameter *keepdims* - (boolean) If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original tensor.

Returns variance of *x* along *axis*

axis can be:

- *None* - in which case the standard deviation is computed along all axes (like numpy)
- an *int* - computed along this axis

- a *list of ints* - computed along these axes

theano.tensor.**all**(*x*, *axis=None*, *keepdims=False*)

Parameter *x* - symbolic Tensor (or compatible)

Parameter *axis* - axis or axes along which to apply 'bitwise and'

Parameter *keepdims* - (boolean) If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original tensor.

Returns bitwise and of *x* along *axis*

axis can be:

- *None* - in which case the 'bitwise and' is computed along all axes (like numpy)
- an *int* - computed along this axis
- a *list of ints* - computed along these axes

theano.tensor.**any**(*x*, *axis=None*, *keepdims=False*)

Parameter *x* - symbolic Tensor (or compatible)

Parameter *axis* - axis or axes along which to apply bitwise or

Parameter *keepdims* - (boolean) If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original tensor.

Returns bitwise or of *x* along *axis*

axis can be:

- *None* - in which case the 'bitwise or' is computed along all axes (like numpy)
- an *int* - computed along this axis
- a *list of ints* - computed along these axes

theano.tensor.**ptp**(*x*, *axis = None*)

Range of values (maximum - minimum) along an axis. The name of the function comes from the acronym for peak to peak.

Parameter *x* Input tensor.

Parameter *axis* Axis along which to find the peaks. By default, flatten the array.

Returns A new array holding the result.

Indexing Like NumPy, Theano distinguishes between *basic* and *advanced* indexing. Theano fully supports basic indexing (see NumPy's indexing).

Integer advanced indexing will be supported in 0.6rc4 (or the development version). We do not support boolean masks, as Theano does not have a boolean type (we use int8 for the output of logic operators).

NumPy with a mask:

```
>>> n = np.arange(9).reshape(3,3)
>>> n[n > 4]
array([5, 6, 7, 8])
```

Theano indexing with a "mask" (incorrect approach):

```
>>> t = theano.tensor.arange(9).reshape((3,3))
>>> t[t > 4].eval()  # an array with shape (3, 3, 3)
array([[[0, 1, 2],
        [0, 1, 2],
        [0, 1, 2]],

       [[0, 1, 2],
        [0, 1, 2],
        [3, 4, 5]],

       [[3, 4, 5],
        [3, 4, 5],
        [3, 4, 5]]], dtype=int8)
```

Getting a Theano result like NumPy:

```
>>> t[(t > 4).nonzero()].eval()
array([5, 6, 7, 8], dtype=int8)
```

The gradient of Advanced indexing needs in many cases NumPy 1.8. It is not released yet as of April 30th, 2013. You can use NumPy development version to have this feature now.

Index-assignment is *not* supported. If you want to do something like `a[5] = b` or `a[5]+=b`, see `theano.tensor.set_subtensor()` and `theano.tensor.inc_subtensor()` below.

`theano.tensor.`**`set_subtensor`**(*x*, *y*, *inplace=False*, *tolerate_inplace_aliasing=False*)

Return x with the given subtensor overwritten by y.

Example: To replicate the numpy expression "r[10:] = 5", type

```
>>> r = ivector()
>>> new_r = set_subtensor(r[10:], 5)
```

Parameters

- **x** – symbolic variable for the lvalue of = operation
- **y** – symbolic variable for the rvalue of = operation
- **tolerate_inplace_aliasing** – see inc_subtensor for documentation.

theano.tensor.**inc_subtensor**(*x*, *y*, *inplace=False*, *set_instead_of_inc=False*, *tolerate_inplace_aliasing=False*)

Return x with the given subtensor incremented by y.

Parameters

- **x** – the symbolic result of a Subtensor operation.
- **y** – the amount by which to increment ths subtensor in question
- **inplace** – Don't use. Theano will do it when possible.
- **set_instead_of_inc** – If True, do a set_subtensor instead.
- **tolerate_inplace_aliasing** – allow x and y to be views of a single underlying array even while working inplace. For correct results, x and y must not be overlapping views; if they overlap, the result of this Op will generally be incorrect. This value has no effect if inplace=False.

Example: To replicate the numpy expression "r[10:] += 5", type

```
>>> r = ivector()
>>> new_r = inc_subtensor(r[10:], 5)
```

Operator Support Many Python operators are supported.

```
>>> a, b = T.itensor3(), T.itensor3() # example inputs
```

Arithmetic

```
>>> a + 3       # T.add(a, 3) -> itensor3
>>> 3 - a       # T.sub(3, a)
>>> a * 3.5     # T.mul(a, 3.5) -> ftensor3 or dtensor3 (depending on casting)
>>> 2.2 / a     # T.truediv(2.2, a)
>>> 2.2 // a    # T.intdiv(2.2, a)
>>> 2.2**a      # T.pow(2.2, a)
>>> b % a       # T.mod(b, a)
```

Bitwise

```
>>> a & b       # T.and_(a,b)    bitwise and (alias T.bitwise_and)
>>> a ^ 1       # T.xor(a,1)     bitwise xor (alias T.bitwise_xor)
>>> a | b       # T.or_(a,b)     bitwise or (alias T.bitwise_or)
>>> ~a          # T.invert(a)    bitwise invert (alias T.bitwise_not)
```

Inplace In-place operators are *not* supported. Theano's graph-optimizations will determine which intermediate values to use for in-place computations. If you would like to update the value of a *shared variable*, consider using the `updates` argument to `theano.function()`.

Elementwise

Casting

theano.tensor.**cast**(*x*, *dtype*)

Cast any tensor *x* to a Tensor of the same shape, but with a different numerical type *dtype*.

This is not a reinterpret cast, but a coersion cast, similar to `numpy.asarray(x, dtype=dtype)`.

```
import theano.tensor as T
x = T.matrix()
x_as_int = T.cast(x, 'int32')
```

Attempting to casting a complex value to a real value is ambiguous and will raise an exception. Use *real()*, *imag()*, *abs()*, or *angle()*.

theano.tensor.**real**(*x*)

Return the real (not imaginary) components of Tensor x. For non-complex *x* this function returns x.

theano.tensor.**imag**(*x*)

Return the imaginary components of Tensor x. For non-complex *x* this function returns zeros_like(x).

Comparisons

The six usual equality and inequality operators share the same interface.

Parameter *a* - symbolic Tensor (or compatible)

Parameter *b* - symbolic Tensor (or compatible)

Return type symbolic Tensor

Returns a symbolic tensor representing the application of the logical elementwise operator.

Note: Theano has no boolean dtype. Instead, all boolean tensors are represented in `'int8'`.

Here is an example with the less-than operator.

```
import theano.tensor as T
x,y = T.dmatrices('x','y')
z = T.le(x,y)
```

theano.tensor.**lt**(*a*, *b*)

Returns a symbolic `'int8'` tensor representing the result of logical less-than (a<b).

Also available using syntax `a < b`

theano.tensor.**gt**(*a*, *b*)

Returns a symbolic `'int8'` tensor representing the result of logical greater-than (a>b).

Also available using syntax `a > b`

theano.tensor.**le**(*a*, *b*)

Returns a variable representing the result of logical less than or equal (a<=b).

Also available using syntax `a <= b`

theano.tensor.**ge**(*a*, *b*)
Returns a variable representing the result of logical greater or equal than (a>=b).

Also available using syntax `a >= b`

theano.tensor.**eq**(*a*, *b*)
Returns a variable representing the result of logical equality (a==b).

theano.tensor.**neq**(*a*, *b*)
Returns a variable representing the result of logical inequality (a!=b).

theano.tensor.**isnan**(*a*)
Returns a variable representing the comparison of `a` elements with nan.

This is equivalent to `numpy.isnan`.

theano.tensor.**isinf**(*a*)
Returns a variable representing the comparison of `a` elements with inf or -inf.

This is equivalent to `numpy.isinf`.

theano.tensor.**isclose**(*a*, *b*, *rtol=1e-05*, *atol=1e-08*, *equal_nan=False*)
Returns a symbolic `'int8'` tensor representing where two tensors are equal within a tolerance.

The tolerance values are positive, typically very small numbers. The relative difference *(rtol * abs(b))* and the absolute difference *atol* are added together to compare against the absolute difference between *a* and *b*.

For finite values, isclose uses the following equation to test whether two floating point values are equivalent: `|a - b| <= (atol + rtol * |b|)`

For infinite values, isclose checks if both values are the same signed inf value.

If equal_nan is True, isclose considers NaN values in the same position to be close. Otherwise, NaN values are not considered close.

This is equivalent to `numpy.isclose`.

theano.tensor.**allclose**(*a*, *b*, *rtol=1e-05*, *atol=1e-08*, *equal_nan=False*)
Returns a symbolic `'int8'` value representing if all elements in two tensors are equal within a tolerance.

See notes in *isclose* for determining values equal within a tolerance.

This is equivalent to `numpy.allclose`.

Condition
theano.tensor.**switch**(*cond*, *ift*, *iff*)

Returns a variable representing a switch between ift (iftrue) and iff (iffalse) based on the condition cond. This is the theano equivalent of numpy.where.

Parameter *cond* - symbolic Tensor (or compatible)

Parameter *ift* - symbolic Tensor (or compatible)

Parameter *iff* - symbolic Tensor (or compatible)

Return type symbolic Tensor

```
import theano.tensor as T
a,b = T.dmatrices('a','b')
x,y = T.dmatrices('x','y')
z = T.switch(T.lt(a,b), x, y)
```

theano.tensor.**where**(*cond*, *ift*, *iff*)
Alias for *switch*. where is the numpy name.

theano.tensor.**clip**(*x*, *min*, *max*)
Return a variable representing x, but with all elements greater than *max* clipped to *max* and all elements less than *min* clipped to *min*.

Normal broadcasting rules apply to each of *x*, *min*, and *max*.

Bit-wise

The bitwise operators possess this interface:

Parameter *a* - symbolic Tensor of integer type.

Parameter *b* - symbolic Tensor of integer type.

Note: The bitwise operators must have an integer type as input.

The bit-wise not (invert) takes only one parameter.

Return type symbolic Tensor with corresponding dtype.

theano.tensor.**and_**(*a*, *b*)
Returns a variable representing the result of the bitwise and.

theano.tensor.**or_**(*a*, *b*)
Returns a variable representing the result of the bitwise or.

theano.tensor.**xor**(*a*, *b*)
Returns a variable representing the result of the bitwise xor.

theano.tensor.**invert**(*a*)
Returns a variable representing the result of the bitwise not.

theano.tensor.**bitwise_and**(*a*, *b*)
Alias for *and_*. bitwise_and is the numpy name.

theano.tensor.**bitwise_or**(*a*, *b*)
Alias for *or_*. bitwise_or is the numpy name.

theano.tensor.**bitwise_xor**(*a*, *b*)
Alias for *xor_*. bitwise_xor is the numpy name.

theano.tensor.**bitwise_not**(*a*, *b*)
Alias for invert. invert is the numpy name.

Here is an example using the bit-wise `and_` via the `&` operator:

```
import theano.tensor as T
x,y = T.imatrices('x','y')
z = x & y
```

Mathematical

theano.tensor.**abs_**(*a*)

Returns a variable representingthe absolute of a, ie |a|.

Note: Can also be accessed with abs(a).

---

theano.tensor.**angle**(*a*)

Returns a variable representing angular component of complex-valued Tensor *a*.

theano.tensor.**exp**(*a*)

Returns a variable representing the exponential of a, ie e^a.

theano.tensor.**maximum**(*a*, *b*)

Returns a variable representing the maximum element by element of a and b

theano.tensor.**minimum**(*a*, *b*)

Returns a variable representing the minimum element by element of a and b

theano.tensor.**neg**(*a*)

Returns a variable representing the negation of *a* (also -a).

theano.tensor.**inv**(*a*)

Returns a variable representing the inverse of a, ie 1.0/a. Also called reciprocal.

theano.tensor.**log**(*a*), *log2(a)*, *log10(a*)

Returns a variable representing the base e, 2 or 10 logarithm of a.

theano.tensor.**sgn**(*a*)

Returns a variable representing the sign of a.

theano.tensor.**ceil**(*a*)

Returns a variable representing the ceiling of a (for example ceil(2.1) is 3).

theano.tensor.**floor**(*a*)

Returns a variable representing the floor of a (for example floor(2.9) is 2).

theano.tensor.**round**(*a*, *mode="half_away_from_zero"*)

Returns a variable representing the rounding of a in the same dtype as a. Implemented rounding mode are half_away_from_zero and half_to_even.

theano.tensor.**iround**(*a*, *mode="half_away_from_zero"*)

Short hand for cast(round(a, mode),'int64').

theano.tensor.**sqr**(*a*)

Returns a variable representing the square of a, ie a^2.

theano.tensor.**sqrt**(*a*)

Returns a variable representing the of a, ie a^0.5.

theano.tensor.**cos**(*a*), *sin(a)*, *tan(a)*
: Returns a variable representing the trigonometric functions of a (cosine, sine and tangent).

theano.tensor.**cosh**(*a*), *sinh(a)*, *tanh(a)*
: Returns a variable representing the hyperbolic trigonometric functions of a (hyperbolic cosine, sine and tangent).

theano.tensor.**erf**(*a*), *erfc(a)*
: Returns a variable representing the error function or the complementary error function. wikipedia

theano.tensor.**erfinv**(*a*), *erfcinv(a)*
: Returns a variable representing the inverse error function or the inverse complementary error function. wikipedia

theano.tensor.**gamma**(*a*)
: Returns a variable representing the gamma function.

theano.tensor.**gammaln**(*a*)
: Returns a variable representing the logarithm of the gamma function.

theano.tensor.**psi**(*a*)
: Returns a variable representing the derivative of the logarithm of the gamma function (also called the digamma function).

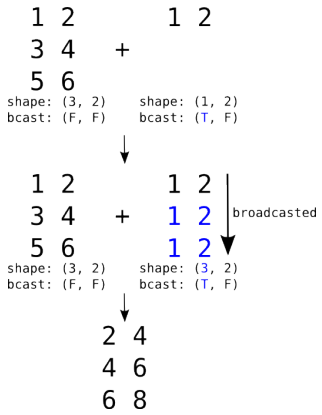
theano.tensor.**chi2sf**(*a*, *df*)
: Returns a variable representing the survival function (1-cdf — sometimes more accurate).

  C code is provided in the Theano_lgpl repository. This makes it faster.

  https://github.com/Theano/Theano_lgpl.git

Broadcasting in Theano vs. Numpy Broadcasting is a mechanism which allows tensors with different numbers of dimensions to be added or multiplied together by (virtually) replicating the smaller tensor along the dimensions that it is lacking.

Broadcasting is the mechanism by which a scalar may be added to a matrix, a vector to a matrix or a scalar to a vector.



Broadcasting a row matrix. T and F respectively stand for True and False and indicate along which dimensions we allow broadcasting.

If the second argument were a vector, its shape would be `(2,)` and its broadcastable pattern `(F,)`. They would be automatically expanded to the **left** to match the dimensions of the matrix (adding `1` to the shape and `T` to the pattern), resulting in `(1, 2)` and `(T, F)`. It would then behave just like the example above.

Unlike numpy which does broadcasting dynamically, Theano needs to know, for any operation which supports broadcasting, which dimensions will need to be broadcasted. When applicable, this information is given in the *Type* of a *Variable*.

See also:

- SciPy documentation about numpy's broadcasting
- OnLamp article about numpy's broadcasting

Linear Algebra

`theano.tensor.`**`dot`**(*X*, *Y*)

Parameters

- **X** (*symbolic matrix or vector*) – left term
- **Y** (*symbolic matrix or vector*) – right term

Return type symbolic matrix or vector

Returns the inner product of *X* and *Y*.

`theano.tensor.`**`outer`**(*X*, *Y*)

Parameters

- **X** (*symbolic vector*) – left term
- **Y** (*symbolic vector*) – right term

Return type symbolic matrix

Returns vector-vector outer product

`theano.tensor.`**`tensordot`**(*a*, *b*, *axes=2*)

Given two tensors a and b,tensordot computes a generalized dot product over the provided axes. Theano's implementation reduces all expressions to matrix or vector dot products and is based on code from Tijmen Tieleman's gnumpy (http://www.cs.toronto.edu/~tijmen/gnumpy.html).

Parameters

- **a** (*symbolic tensor*) – the first tensor variable
- **b** (*symbolic tensor*) – the second tensor variable
- **axes** (*int or array-like of length 2*) – an integer or array. If an integer, the number of axes to sum over. If an array, it must have two array elements containing the axes to sum over in each tensor.

  Note that the default value of 2 is not guaranteed to work for all values of a and b, and an error will be raised if that is the case. The reason for keeping the default is to maintain the same signature as numpy's tensordot function (and np.tensordot raises analogous errors for non-compatible inputs).

If an integer i, it is converted to an array containing the last i dimensions of the first tensor and the first i dimensions of the second tensor:

axes = [range(a.ndim - i, b.ndim), range(i)]

If an array, its two elements must contain compatible axes of the two tensors. For example, [[1, 2], [2, 0]] means sum over the 2nd and 3rd axes of a and the 3rd and 1st axes of b. (Remember axes are zero-indexed!) The 2nd axis of a and the 3rd axis of b must have the same shape; the same is true for the 3rd axis of a and the 1st axis of b.

Returns a tensor with shape equal to the concatenation of a's shape (less any dimensions that were summed over) and b's shape (less any dimensions that were summed over).

Return type symbolic tensor

It may be helpful to consider an example to see what tensordot does. Theano's implementation is identical to NumPy's. Here a has shape (2, 3, 4) and b has shape (5, 6, 4, 3). The axes to sum over are [[1, 2], [3, 2]] – note that a.shape[1] == b.shape[3] and a.shape[2] == b.shape[2]; these axes are compatible. The resulting tensor will have shape (2, 5, 6) – the dimensions that are not being summed:

```
import numpy as np

a = np.random.random((2,3,4))
b = np.random.random((5,6,4,3))

#tensordot
c = np.tensordot(a, b, [[1,2],[3,2]])

#loop replicating tensordot
a0, a1, a2 = a.shape
b0, b1, _, _ = b.shape
cloop = np.zeros((a0,b0,b1))

#loop over non-summed indices -- these exist
#in the tensor product.
for i in range(a0):
    for j in range(b0):
        for k in range(b1):
            #loop over summed indices -- these don't exist
            #in the tensor product.
            for l in range(a1):
                for m in range(a2):
                    cloop[i,j,k] += a[i,l,m] * b[j,k,m,l]

assert np.allclose(c, cloop)
```

This specific implementation avoids a loop by transposing a and b such that the summed axes of a are last and the summed axes of b are first. The resulting arrays are reshaped to 2 dimensions (or left as vectors, if appropriate) and a matrix or vector dot product is taken. The result is reshaped back to the required output dimensions.

In an extreme case, no axes may be specified. The resulting tensor will have shape equal to the concatenation of the shapes of a and b:

```
>>> c = np.tensordot(a, b, 0)
>>> a.shape
(2, 3, 4)
>>> b.shape
(5, 6, 4, 3)
>>> print(c.shape)
(2, 3, 4, 5, 6, 4, 3)
```

Note See the documentation of numpy.tensordot for more examples.

theano.tensor.**batched_dot**(*X*, *Y*)

Parameters

- **x** – A Tensor with sizes e.g.: for 3D (dim1, dim3, dim2)
- **y** – A Tensor with sizes e.g.: for 3D (dim1, dim2, dim4)

This function computes the dot product between the two tensors, by iterating over the first dimension using scan. Returns a tensor of size e.g. if it is 3D: (dim1, dim3, dim4) Example:

```
>>> first = T.tensor3('first')
>>> second = T.tensor3('second')
>>> result = batched_dot(first, second)
```

Note This is a subset of numpy.einsum, but we do not provide it for now. But numpy einsum is slower than dot or tensordot: http://mail.scipy.org/pipermail/numpy-discussion/2012-October/064259.html

Parameters

- **X** (*symbolic tensor*) – left term
- **Y** (*symbolic tensor*) – right term

Returns tensor of products

theano.tensor.**batched_tensordot**(*X*, *Y*, *axes=2*)

Parameters

- **x** – A Tensor with sizes e.g.: for 3D (dim1, dim3, dim2)
- **y** – A Tensor with sizes e.g.: for 3D (dim1, dim2, dim4)
- **axes** (*int or array-like of length 2*) – an integer or array. If an integer, the number of axes to sum over. If an array, it must have two array elements containing the axes to sum over in each tensor.

  If an integer i, it is converted to an array containing the last i dimensions of the first tensor and the first i dimensions of the second tensor (excluding the first (batch) dimension):

  ```
  axes = [range(a.ndim - i, b.ndim), range(1,i+1)]
  ```

If an array, its two elements must contain compatible axes of the two tensors. For example, [[1, 2], [2, 4]] means sum over the 2nd and 3rd axes of a and the 3rd and 5th axes of b. (Remember axes are zero-indexed!) The 2nd axis of a and the 3rd axis of b must have the same shape; the same is true for the 3rd axis of a and the 5th axis of b.

Returns a tensor with shape equal to the concatenation of a's shape (less any dimensions that were summed over) and b's shape (less first dimension and any dimensions that were summed over).

Return type tensor of tensordots

A hybrid of batch_dot and tensordot, this function computes the tensordot product between the two tensors, by iterating over the first dimension using scan to perform a sequence of tensordots.

Note See `tensordot()` and `batched_dot()` for supplementary documentation.

`theano.tensor.`**`mgrid`**`()`

Returns an instance which returns a dense (or fleshed out) mesh-grid when indexed, so that each returned argument has the same shape. The dimensions and number of the output arrays are equal to the number of indexing dimensions. If the step length is not a complex number, then the stop is not inclusive.

Example:

```
>>> a = T.mgrid[0:5, 0:3]
>>> a[0].eval()
array([[0, 0, 0],
       [1, 1, 1],
       [2, 2, 2],
       [3, 3, 3],
       [4, 4, 4]], dtype=int8)
>>> a[1].eval()
array([[0, 1, 2],
       [0, 1, 2],
       [0, 1, 2],
       [0, 1, 2],
       [0, 1, 2]], dtype=int8)
```

`theano.tensor.`**`ogrid`**`()`

Returns an instance which returns an open (i.e. not fleshed out) mesh-grid when indexed, so that only one dimension of each returned array is greater than 1. The dimension and number of the output arrays are equal to the number of indexing dimensions. If the step length is not a complex number, then the stop is not inclusive.

Example:

```
>>> b = T.ogrid[0:5, 0:3]
>>> b[0].eval()
array([[0],
       [1],
       [2],
       [3],
```

```
       [4]], dtype=int8)
>>> b[1].eval()
array([[0, 1, 2, 3]], dtype=int8)
```

Gradient / Differentiation Driver for gradient calculations.

theano.gradient.**grad**(*cost*, *wrt*, *consider_constant=None*, *disconnected_inputs='raise'*, *add_names=True*, *known_grads=None*, *return_disconnected='zero'*)

Return symbolic gradients for one or more variables with respect to some cost.

For more information about how automatic differentiation works in Theano, see `gradient`. For information on how to implement the gradient of a certain Op, see `grad()`.

Parameters

- **cost** (*Scalar (0-dimensional) tensor variable. May optionally be None if known_grads is provided.*) – a scalar with respect to which we are differentiating
- **wrt** (*Tensor variable or list of variables.*) – term[s] for which we want gradients
- **consider_constant** (*list of variables*) – a list of expressions not to backpropagate through
- **disconnected_inputs** (*string*) – Defines the behaviour if some of the variables in `wrt` are not part of the computational graph computing `cost` (or if all links are non-differentiable). The possible values are: - 'ignore': considers that the gradient on these parameters is zero. - 'warn': consider the gradient zero, and print a warning. - 'raise': raise DisconnectedInputError.
- **add_names** (*bool*) – If True, variables generated by grad will be named (d<cost.name>/d<wrt.name>) provided that both cost and wrt have names
- **known_grads** (*dict*) – If not None, a dictionary mapping variables to their gradients. This is useful in the case where you know the gradient on some variables but do not know the original cost.
- **return_disconnected** (*string*) –
  - **'zero'** [If wrt[i] is disconnected, return value i will be] wrt[i].zeros_like()
  - **'None'** [If wrt[i] is disconnected, return value i will be] None
  - 'Disconnected' : returns variables of type DisconnectedType

Return type variable or list/tuple of Variables (matching *wrt*)

Returns symbolic expression of gradient of *cost* with respect to each of the *wrt* terms. If an element of *wrt* is not differentiable with respect to the output, then a zero variable is returned. It returns an object of same type as *wrt*: a list/tuple or Variable in all cases.

See the *gradient* page for complete documentation of the gradient module.

List of Implemented R op See the *gradient tutorial* for the R op documentation.

list of ops that support R-op:

- **with test [Most is tensor/tests/test_rop.py]**
  - SpecifyShape
  - MaxAndArgmax
  - Subtensor
  - IncSubtensor set_subtensor too
  - Alloc
  - Dot
  - Elemwise
  - Sum
  - Softmax
  - Shape
  - Join
  - Rebroadcast
  - Reshape
  - Flatten
  - DimShuffle
  - Scan [In scan_module/tests/test_scan.test_rop]
- **without test**
  - Split
  - ARange
  - ScalarFromTensor
  - AdvancedSubtensor1
  - AdvancedIncSubtensor1
  - AdvancedIncSubtensor

Partial list of ops without support for R-op:

- All sparse ops
- All linear algebra ops.
- PermuteRowElements
- Tile
- AdvancedSubtensor

- TensorDot
- Outer
- Prod
- MulwithoutZeros
- ProdWithoutZeros
- CAReduce(for max,... done for MaxAndArgmax op)
- MaxAndArgmax(only for matrix on axis 0 or 1)

`nnet` – Ops related to neural networks

Theano was originally developped for machine learning applications, particularly for the topic of deep learning. As such, our lab has developed many functions and ops which are particular to neural networks and deep learning.

`conv` – Ops for convolutional neural nets

Note: Two similar implementation exists for conv2d:

`signal.conv2d` and `nnet.conv2d`.

The former implements a traditional 2D convolution, while the latter implements the convolutional layers present in convolutional neural networks (where filters are 3D and pool over several input channels).

Note: As of October 21st, 2014, the default GPU image convolution changed: By default, if *cuDNN* is available, we will use it, otherwise we will fall back to using the gemm version (slower then cuDNN in most cases and uses more memory).

Both cuDNN and the gemm version can be disabled using the Theano flags `optimizer_excluding=conv_dnn` and `optimizer_excluding=conv_gemm`, respectively. In this case, we will fall back to using the legacy convolution code, which is slower, but does not require extra memory. To verify that cuDNN is used, you can supply the Theano flag `optimizer_including=cudnn`. This will raise an error if cuDNN is unavailable.

It is not advised to ever disable cuDNN, as this is usually the fastest option. Disabling the gemm version is only useful if cuDNN is unavailable and you run out of GPU memory.

There are two other implementations: An FFT-based convolution integrated into Theano, and an implementation by Alex Krizhevsky available via Pylearn2. See the documentation below on how to use them.

As of November 24th, 2014, you can also use a meta-optimizer to automatically choose the fastest implementation for each specific convolution in your graph. For each instance, it will compile and benchmark each applicable implementation of the ones listed below and choose the fastest one. As performance is dependent on input and filter shapes, this only works for operations introduced via nnet.conv2d with fully specified shape information. Enable it via the Theano flag `optimizer_including=conv_meta`, and optionally set it to verbose mode via the flag *metaopt.verbose=1*.

TODO: Give examples on how to use these things! They are pretty complicated.

- **Implemented operators for neural network 2D / image convolution:**
  - `nnet.conv2d`. This is the standard operator for convolutional neural networks working with batches of multi-channel 2D images, available for CPU and GPU. It computes a convolution, i.e., it flips the kernel. Most of the more efficient GPU implementations listed below can be inserted automatically as a replacement for nnet.conv2d via graph optimizations. Some of these graph optimizations are enabled by default, others can be enabled via Theano flags.
  - `conv2d_fft` This is a GPU-only version of nnet.conv2d that uses an FFT transform to perform the work. It flips the kernel just like `conv2d`. conv2d_fft should not be used directly as it does not provide a gradient. Instead, use nnet.conv2d and allow Theano's graph optimizer to replace it by the FFT version by setting 'THEANO_FLAGS=optimizer_including=conv_fft' in your environment. If enabled, it will take precedence over cuDNN and the gemm version. It is not enabled by default because it has some restrictions on input and uses a lot more memory. Also note that it requires CUDA >= 5.0, scikits.cuda >= 0.5.0 and PyCUDA to run. To deactivate the FFT optimization on a specific nnet.conv2d while the optimization flag is active, you can set its `version` parameter to `'no_fft'`. To enable it for just one Theano function:

    ```
    mode = theano.compile.get_default_mode()
    mode = mode.including('conv_fft')

    f = theano.function(..., mode=mode)
    ```

  - cuda-convnet wrapper for 2d correlation

    Wrapper for an open-source GPU-only implementation of conv2d by Alex Krizhevsky, very fast, but with several restrictions on input and kernel shapes, and with a different memory layout for the input. It does not flip the kernel.

    This is in Pylearn2, where it is normally called from the linear transform implementation, but it can also be used directly from within Theano as a manual replacement for nnet.conv2d.

  - `GpuCorrMM` This is a GPU-only 2d correlation implementation taken from caffe and also used by Torch. It does not flip the kernel.

    For each element in a batch, it first creates a Toeplitz matrix in a CUDA kernel. Then, it performs a `gemm` call to multiply this Toeplitz matrix and the filters (hence the name: MM is for matrix multiplication). It needs extra memory for the Toeplitz matrix, which is a 2D matrix of shape `(no of channels * filter width * filter height, output width * output height)`.

    As it provides a gradient, you can use it as a replacement for nnet.conv2d. But usually, you will just use nnet.conv2d and allow Theano's graph optimizer to automatically replace it by the GEMM version if cuDNN is not available. To explicitly disable the graph optimizer, set `THEANO_FLAGS=optimizer_excluding=conv_gemm` in your environment. If using it, please see the warning about a bug in CUDA 5.0 to 6.0 below.

  - `dnn_conv` GPU-only convolution using NVIDIA's cuDNN library. This requires that you have cuDNN installed and available, which in turn requires CUDA 6.5 and a GPU with

compute capability 3.0 or more.

If cuDNN is available, by default, Theano will replace all nnet.conv2d operations with dnn_conv. To explicitly disable it, set `THEANO_FLAGS=optimizer_excluding=conv_dnn` in your environment. As dnn_conv has a gradient defined, you can also use it manually.

- **Implemented operators for neural network 3D / video convolution:**
  - `conv3D` 3D Convolution applying multi-channel 3D filters to batches of multi-channel 3D images. It does not flip the kernel.
  - `conv3d_fft` GPU-only version of conv3D using FFT transform. conv3d_fft should not be called directly as it does not provide a gradient. Instead, use conv3D and allow Theano's graph optimizer to replace it by the FFT version by setting `THEANO_FLAGS=optimizer_including=conv3d_fft:convgrad3d_fft:convtransp3d_` in your environment. This is not enabled by default because it does not support strides and uses more memory. Also note that it requires CUDA >= 5.0, scikits.cuda >= 0.5.0 and PyCUDA to run. To enable for just one Theano function:

    ```
    mode = theano.compile.get_default_mode()
    mode = mode.including('conv3d_fft', 'convgrad3d_fft', 'convtransp3d_fft')

    f = theano.function(..., mode=mode)
    ```

  - `GpuCorr3dMM` This is a GPU-only 3d correlation relying on a Toeplitz matrix and gemm implementation (see `GpuCorrMM`) It needs extra memory for the Toeplitz matrix, which is a 2D matrix of shape `(no of channels * filter width * filter height * filter depth, output width * output height * output depth)`. As it provides a gradient, you can use it as a replacement for nnet.conv3d. Alternatively, you can use nnet.conv3d and allow Theano's graph optimizer to replace it by the GEMM version by setting `THEANO_FLAGS=optimizer_including=conv3d_gemm:convgrad3d_gemm:convtransp3` in your environment. This is not enabled by default because it uses some extra memory, but the overhead is small compared to conv3d_fft, there are no restrictions on input or kernel shapes and strides are supported. If using it, please see the warning about a bug in CUDA 5.0 to 6.0 in `GpuCorrMM`.
  - `conv3d2d` Another conv3d implementation that uses the conv2d with data reshaping. It is faster in some cases than conv3d, and work on the GPU. It flip the kernel.

`theano.tensor.nnet.conv.`**`conv2d`**(*input*, *filters*, *image_shape=None*, *filter_shape=None*, *border_mode='valid'*, *subsample=(1, 1)*, ***kargs*)

This function will build the symbolic graph for convolving a stack of input images with a set of filters. The implementation is modelled after Convolutional Neural Networks (CNN). It is simply a wrapper to the ConvOp but provides a much cleaner interface.

Parameters

- **input** (*symbolic 4D tensor*) – mini-batch of feature map stacks, of shape (batch size, stack size, nb row, nb col) see the optional parameter image_shape

- **filters** (*symbolic 4D tensor*) – set of filters used in CNN layer of shape (nb filters, stack size, nb row, nb col) see the optional parameter filter_shape
- **border_mode** –

  'valid'– only apply filter to complete patches of the image. Generates output of shape: image_shape - filter_shape + 1

  'full' – zero-pads image to multiple of filter shape to generate output of shape: image_shape + filter_shape - 1
- **subsample** (*tuple of len 2*) – factor by which to subsample the output. Also called strides elsewhere.
- **image_shape** (*None, tuple/list of len 4 of int, None or Constant variable*) – The shape of the input parameter. Optional, used for optimization like loop unrolling You can put None for any element of the list to tell that this element is not constant.
- **filter_shape** (*None, tuple/list of len 4 of int, None or Constant variable*) – Optional, used for optimization like loop unrolling You can put None for any element of the list to tell that this element is not constant.
- **kwargs** – kwargs are passed onto ConvOp. Can be used to set the following: unroll_batch, unroll_kern, unroll_patch, openmp (see ConvOp doc)

  openmp: By default have the same value as config.openmp. For small image, filter, batch size, nkern and stack size, it can be faster to disable manually openmp. A fast and incomplete test show that with image size 6x6, filter size 4x4, batch size==1, n kern==1 and stack size==1, it is faster to disable it in valid mode. But if we grow the batch size to 10, it is faster with openmp on a core 2 duo.

Return type symbolic 4D tensor

Returns set of feature maps generated by convolutional layer. Tensor is of shape (batch size, nb filters, output row, output col)

`theano.sandbox.cuda.fftconv.`**`conv2d_fft`**(*input*, *filters*, *image_shape=None*, *filter_shape=None*, *border_mode='valid'*, *pad_last_dim=False*)

Perform a convolution through fft.

Only support input which will be even on the last dimension (width). All other dimensions can be anything and the filters can have an even or odd width.

If you must use input which has an odd width, you can either pad it or use the *pad_last_dim* argument which will do it for you and take care to strip the padding before returning. Don't use this argument if you are not sure the input is odd since the padding is unconditional and will make even input odd, thus leading to problems.

On valid mode the filters must be smaller than the input.

input (b, ic, i0, i1).

filters (oc, ic, f0, f1).

border_mode : {'valid', 'full'} pad_last_dim

Unconditionally pad the last dimension of the input to to turn it from odd to even. Will strip the padding before returning the result.

theano.tensor.nnet.Conv3D.**conv3D**(*V*, *W*, *b*, *d*)

3D "convolution" of multiple filters on a minibatch (does not flip the kernel, moves kernel with a user specified stride)

Parameters

- **V** – Visible unit, input. dimensions: (batch, row, column, time, in channel)
- **W** – Weights, filter. dimensions: (out channel, row, column, time ,in channel)
- **b** – bias, shape == (W.shape[0],)
- **d** – strides when moving the filter over the input(dx, dy, dt)

Note The order of dimensions does not correspond to the one in *conv2d*. This is for optimization.

Note The GPU implementation is very slow. You should use conv3d2d or conv3d_fft for a GPU graph instead.

See Someone made a script that shows how to swap the axes between both 3d convolution implementations in Theano. See the last attachment.

theano.sandbox.cuda.fftconv.**conv3d_fft**(*input*, *filters*, *image_shape=None*, *filter_shape=None*, *border_mode='valid'*, *pad_last_dim=False*)

Perform a convolution through fft.

Only supports input whose shape is even on the last dimension. All other dimensions can be anything and the filters can have an even or odd last dimension.

The semantics associated with the last three dimensions are not important as long as they are in the same order between the inputs and the filters. For example, when the convolution is done on a sequence of images, they could be either (duration, height, width) or (height, width, duration).

If you must use input which has an odd width, you can either pad it or use the *pad_last_dim* argument which will do it for you and take care to strip the padding before returning. pad_last_dim checks that the last dimension is odd before the actual paddding

On valid mode the filters must be smaller than the input.

input (b, ic, i0, i1, i2).

filters (oc, ic, f0, f1, i2).

border_mode : {'valid', 'full'}. pad_last_dim

Unconditionally pad the last dimension of the input to to turn it from odd to even. Will strip the padding before returning the result.

theano.tensor.nnet.conv3d2d.**conv3d**(*signals*, *filters*, *signals_shape=None*, *filters_shape=None*, *border_mode='valid'*)

Convolve spatio-temporal filters with a movie.

It flips the filters.

Parameters

- **signals** – timeseries of images whose pixels have color channels. shape: [Ns, Ts, C, Hs, Ws]
- **filters** – spatio-temporal filters shape: [Nf, Tf, C, Hf, Wf]
- **signals_shape** – None or a tuple/list with the shape of signals
- **filters_shape** – None or a tuple/list with the shape of filters
- **border_mode** – The only one tested is 'valid'.

Note Another way to define signals: (batch, time, in channel, row, column) Another way to define filters: (out channel,time,in channel, row, column)

Note For the GPU, you can use this implementation or `conv3d_fft`.

See Someone made a script that shows how to swap the axes between both 3d convolution implementations in Theano. See the last attachment.

`nnet` – Ops for neural networks

- **Sigmoid**
  - `sigmoid()`
  - `ultra_fast_sigmoid()`
  - `hard_sigmoid()`
- **Others**
  - `softplus()`
  - `softmax()`
  - `relu()`
  - `binary_crossentropy()`
  - `categorical_crossentropy()`

`tensor.nnet.`**`sigmoid`**`(`x`)`

Returns the standard sigmoid nonlinearity applied to x

Parameters x - symbolic Tensor (or compatible)

Return type same as x

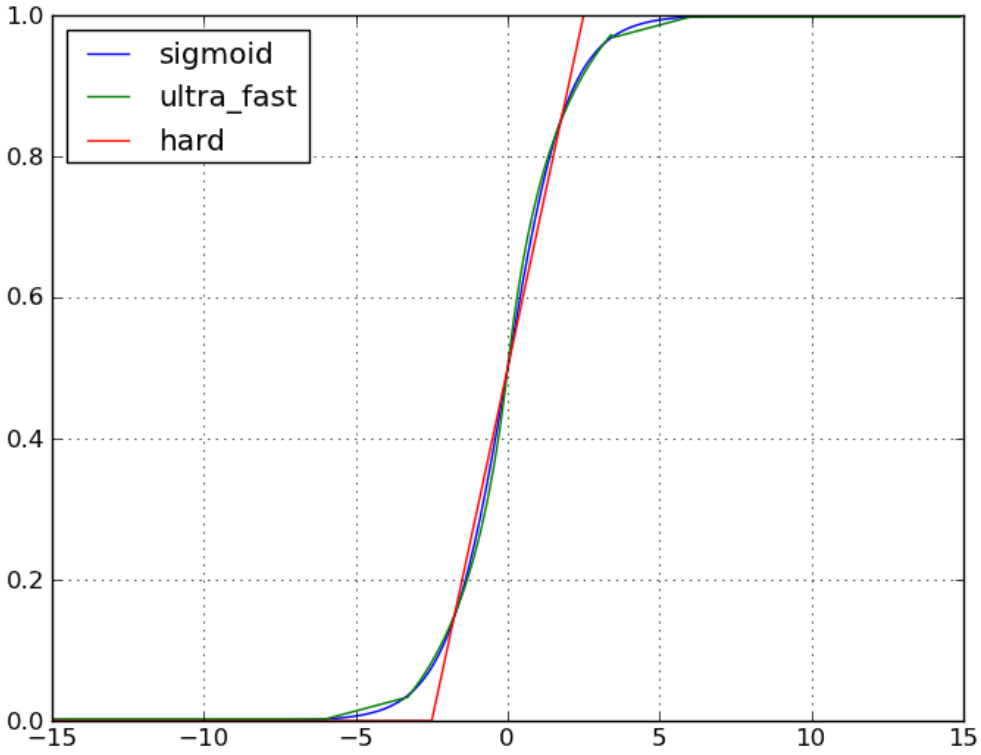
Returns element-wise sigmoid: $sigmoid(x) = \frac{1}{1+\exp(-x)}$.

note see `ultra_fast_sigmoid()` or `hard_sigmoid()` for faster versions. Speed comparison for 100M float64 elements on a Core2 Duo @ 3.16 GHz:

- hard_sigmoid: 1.0s

- ultra_fast_sigmoid: 1.3s
- sigmoid (with amdlibm): 2.3s
- sigmoid (without amdlibm): 3.7s

Precision: sigmoid(without or without amdlibm) > ultra_fast_sigmoid > hard_sigmoid.



Example:

```
x,y,b = T.dvectors('x','y','b')
W = T.dmatrix('W')
y = T.nnet.sigmoid(T.dot(W,x) + b)
```

Note: The underlying code will return an exact 0 or 1 if an element of x is too small or too big.

`tensor.nnet.`**`ultra_fast_sigmoid`**`(`x`)`

Returns the *approximated* standard `sigmoid()` nonlinearity applied to x.

Parameters x - symbolic Tensor (or compatible)

Return type same as x

Returns approximated element-wise sigmoid: $sigmoid(x) = \frac{1}{1+\exp(-x)}$.

note To automatically change all `sigmoid()` ops to this version, use the Theano optimization `local_ultra_fast_sigmoid`. This can be done with the Theano flag `optimizer_including=local_ultra_fast_sigmoid`. This optimization is done late, so it should not affect stabilization optimization.

Note: The underlying code will return 0.00247262315663 as the minimum value and 0.997527376843 as the maximum value. So it never returns 0 or 1.

Note: Using directly the ultra_fast_sigmoid in the graph will disable stabilization optimization associated with it. But using the optimization to insert them won't disable the stability optimization.

`tensor.nnet.`**`hard_sigmoid`**`(`x`)`

Returns the *approximated* standard `sigmoid()` nonlinearity applied to x.

Parameters x - symbolic Tensor (or compatible)

Return type same as x

Returns approximated element-wise sigmoid: $sigmoid(x) = \frac{1}{1+\exp(-x)}$.

note To automatically change all `sigmoid()` ops to this version, use the Theano optimization `local_hard_sigmoid`. This can be done with the Theano flag `optimizer_including=local_hard_sigmoid`. This optimization is done late, so it should not affect stabilization optimization.

Note: The underlying code will return an exact 0 or 1 if an element of x is too small or too big.

Note: Using directly the ultra_fast_sigmoid in the graph will disable stabilization optimization associated with it. But using the optimization to insert them won't disable the stability optimization.

`tensor.nnet.`**`softplus`**`(`x`)`

Returns the softplus nonlinearity applied to x

Parameter x - symbolic Tensor (or compatible)

Return type same as x

Returns elementwise softplus: $softplus(x) = \log_e(1 + \exp(x))$.

Note: The underlying code will return an exact 0 if an element of x is too small.

```
x,y,b = T.dvectors('x','y','b')
W = T.dmatrix('W')
y = T.nnet.softplus(T.dot(W,x) + b)
```

`tensor.nnet.`**`softmax`**`(`x`)`

Returns the softmax function of x:

Parameter x symbolic **2D** Tensor (or compatible).

Return type same as x

Returns a symbolic 2D tensor whose ijth element is $softmax_{ij}(x) = \frac{\exp x_{ij}}{\sum_k \exp(x_{ik})}$.

The softmax function will, when applied to a matrix, compute the softmax values row-wise.

note this insert a particular op. But this op don't yet implement the Rop for hessian free. If you want that, implement this equivalent code that have the Rop implemented `exp(x)/exp(x).sum(1, keepdims=True)`. Theano should optimize this by inserting the softmax op itself. The code of the softmax op is more numeriacaly stable by using this code:

```
e_x = exp(x - x.max(axis=1, keepdims=True))
out = e_x / e_x.sum(axis=1, keepdims=True)
```

Example of use:

```
x,y,b = T.dvectors('x','y','b')
W = T.dmatrix('W')
y = T.nnet.softmax(T.dot(W,x) + b)
```

`theano.tensor.nnet.`**`relu`**(*x*, *alpha=0*)

Compute the element-wise rectified linear activation function.

Parameters

- **x** (*symbolic tensor*) – Tensor to compute the activation function for.
- **alpha** (*scalar or tensor, optional*) – Slope for negative input, usually between 0 and 1. The default value of 0 will lead to the standard rectifier, 1 will lead to a linear activation function, and any value in between will give a leaky rectifier. A shared variable (broadcastable against *x*) will result in a parameterized rectifier with learnable slope(s).

Return type symbolic tensor

Returns element-wise rectifier applied to *x*

Note: This is numerically equivalent to `T.switch(x > 0, x, alpha * x)` (or `T.maximum(x, alpha * x)` for `alpha < 1`), but uses a faster formulation or an optimized Op, so we encourage to use this function.

`tensor.nnet.`**`binary_crossentropy`**(*output*, *target*)

Computes the binary cross-entropy between a target and an output:

Parameters

- *target* - symbolic Tensor (or compatible)
- *output* - symbolic Tensor (or compatible)

Return type same as target

Returns a symbolic tensor, where the following is applied elementwise $crossentropy(t, o) = -(t \cdot log(o) + (1 - t) \cdot log(1 - o))$.

The following block implements a simple auto-associator with a sigmoid nonlinearity and a reconstruction error which corresponds to the binary cross-entropy (note that this assumes that x will contain values between 0 and 1):

```
x, y, b = T.dvectors('x', 'y', 'b')
W = T.dmatrix('W')
h = T.nnet.sigmoid(T.dot(W, x) + b)
x_recons = T.nnet.sigmoid(T.dot(V, h) + c)
recon_cost = T.nnet.binary_crossentropy(x_recons, x).mean()
```

tensor.nnet.**categorical_crossentropy**(*coding_dist*, *true_dist*)

Return the cross-entropy between an approximating distribution and a true distribution. The cross entropy between two probability distributions measures the average number of bits needed to identify an event from a set of possibilities, if a coding scheme is used based on a given probability distribution q, rather than the "true" distribution p. Mathematically, this function computes $H(p,q) = -\sum_x p(x)\log(q(x))$, where p=true_dist and q=coding_dist.

Parameters

- *coding_dist* - symbolic 2D Tensor (or compatible). Each row represents a distribution.
- *true_dist* - symbolic 2D Tensor **OR** symbolic vector of ints. In the case of an integer vector argument, each element represents the position of the '1' in a 1-of-N encoding (aka "one-hot" encoding)

Return type tensor of rank one-less-than *coding_dist*

Note: An application of the scenario where *true_dist* has a 1-of-N representation is in classification with softmax outputs. If *coding_dist* is the output of the softmax and *true_dist* is a vector of correct labels, then the function will compute `y_i = - \log(coding_dist[i, one_of_n[i]])`, which corresponds to computing the neg-log-probability of the correct class (which is typically the training criterion in classification settings).

```
y = T.nnet.softmax(T.dot(W, x) + b)
cost = T.nnet.categorical_crossentropy(y, o)
# o is either the above-mentioned 1-of-N vector or 2D tensor
```

neighbours – Ops for working with images in convolutional nets

Functions

theano.tensor.nnet.neighbours.**images2neibs**(*ten4*, *neib_shape*, *neib_step=None*, *mode='valid'*)

Function images2neibs allows to apply a sliding window operation to a tensor containing images or other two-dimensional objects. The sliding window operation loops over points in input data and stores a rectangular neighbourhood of each point. It is possible to assign a step of selecting patches (parameter *neib_step*).

Parameters

- **ten4** (*A 4d tensor-like.*) – A 4-dimensional tensor which represents a list of lists of images.a list of lists of images. It should have shape (list 1 dim, list 2 dim, row, col). The first two dimensions can be useful to store different channels and batches.
- **neib_shape** (*A 1d tensor-like of 2 values.*) – A tuple containing two values: height and width of the neighbourhood. It should have shape (r,c) where r is the height of the neighborhood in rows and c is the width of the neighborhood in columns
- **neib_step** (*A 1d tensor-like of 2 values.*) – (dr,dc) where dr is the number of rows to skip between patch and dc is the number of columns. The parameter should be a tuple of two elements: number of rows and number of columns to skip each iteration. Basically, when the step is 1, the neighbourhood of every first element is taken and every possible rectangular subset is returned. By default it is equal to *neib_shape* in other words, the patches are disjoint. When the step is greater than *neib_shape*, some elements are omitted. When None, this is the same as neib_shape(patch are disjoint)

Note: Currently the step size should be chosen in the way that the corresponding dimension i (width or height) is equal to $n * step\_size_i + neib\_shape_i$ for some n

---

- **mode** (*str*) – Possible values:

  valid Requires an input that is a multiple of the pooling factor (in each direction)

  ignore_borders Same as valid, but will ignore the borders if the shape(s) of the input is not a multiple of the pooling factor(s)

  wrap_centered ?? TODO comment

Returns

Reshapes the input as a 2D tensor where each row is an pooling example. Pseudo-code of the output:

```
idx = 0
for i in xrange(list 1 dim):
    for j in xrange(list 2 dim):
        for k in <image column coordinates>:
            for l in <image row coordinates>:
                output[idx,:]
                     = flattened version of ten4[i,j,l:l+r,k:k+c]
                idx += 1
```

Note: The operation isn't necessarily implemented internally with these for loops, they're just the easiest way to describe the output pattern.

Example:

```
# Defining variables
images = T.tensor4('images')
neibs = images2neibs(images, neib_shape=(5, 5))

# Constructing theano function
window_function = theano.function([images], neibs)

# Input tensor (one image 10x10)
im_val = np.arange(100.).reshape((1, 1, 10, 10))

# Function application
neibs_val = window_function(im_val)
```

Note: The underlying code will construct a 2D tensor of disjoint patches 5x5. The output has shape 4x25.

`theano.tensor.nnet.neighbours.`**`neibs2images`**(*neibs*, *neib_shape*, *original_shape*, *mode='valid'*)

Function `neibs2images` performs the inverse operation of `images2neibs`. It inputs the output of `images2neibs` and reconstructs its input.

Parameters

- **neibs** – matrix like the one obtained by `images2neibs`
- **neib_shape** – *neib_shape* that was used in `images2neibs`
- **original_shape** – original shape of the 4d tensor given to `images2neibs`

Returns Reconstructs the input of `images2neibs`, a 4d tensor of shape *original_shape*.

Note: Currently, the function doesn't support tensors created with *neib_step* different from default value. This means that it may be impossible to compute the gradient of a variable gained by `images2neibs` w.r.t. its inputs in this case, because it uses `images2neibs` for gradient computation.

Example, which uses a tensor gained in example for `images2neibs`:

```
im_new = neibs2images(neibs, (5, 5), im_val.shape)
# Theano function definition
inv_window = theano.function([neibs], im_new)
# Function application
im_new_val = inv_window(neibs_val)
```

Note: The code will output the initial image array.

See also

- *Indexing*
- *scan – Looping in Theano*

raw_random – Low-level random numbers

Raw random provides the random-number drawing functionality, that underlies the friendlier `RandomStreams` interface.

Reference

class `raw_random.`**RandomStreamsBase**(*object*)

This is the interface for the `theano.tensor.shared_randomstreams.RandomStreams` subclass

binomial(self, size=(), n=1, p=0.5, ndim=None):

Sample `n` times with probability of success `p` for each trial and return the number of successes.

If `size` is ambiguous on the number of dimensions, `ndim` may be a plain integer to supplement the missing information.

This wraps the numpy implementation, so it has the same behavior.

uniform(self, size=(), low=0.0, high=1.0, ndim=None):

Sample a tensor of the given size whose elements come from a uniform distribution between low and high.

If `size` is ambiguous on the number of dimensions, `ndim` may be a plain integer to supplement the missing information.

This wraps the numpy implementation, so it has the same bounds: [`low`, `high`[.

normal(self, size=(), avg=0.0, std=1.0, ndim=None):

Sample from a normal distribution centered on `avg` with the specified standard deviation (`std`)

If `size` is ambiguous on the number of dimensions, `ndim` may be a plain integer to supplement the missing information.

This wrap numpy implementation, so it have the same behavior.

random_integers(self, size=(), low=0, high=1, ndim=None):

Sample a random integer between low and high, both inclusive.

If `size` is ambiguous on the number of dimensions, `ndim` may be a plain integer to supplement the missing information.

This is a generalization of `numpy.random.random_integers()` to the case where low and high are tensors. Otherwise it behaves the same.

choice(self, size=(), a=2, replace=True, p=None, ndim=None, dtype='int64'):

Choose values from `a` with or without replacement. `a` can be a 1-D array or a positive scalar. If `a` is a scalar, the samples are drawn from the range [0, `a`[.

If `size` is ambiguous on the number of dimensions, `ndim` may be a plain integer to supplement the missing information.

This wraps the numpy implementation so it has the same behavior.

poisson(self, size=(), lam=None, ndim=None, dtype='int64'):

Draw samples from a Poisson distribution.

The Poisson distribution is the limit of the Binomial distribution for large N.

If `size` is ambiguous on the number of dimensions, `ndim` may be a plain integer to supplement the missing information.

This wraps the numpy implementation so it has the same behavior.

permutation(self, size=(), n=1, ndim=None):
Returns permutations of the integers between 0 and `n-1`, as many times as required by `size`. For instance, if `size=(p,q)`, `p*q` permutations will be generated, and the output shape will be `(p,q,n)`, because each permutation is of size `n`.

Theano tries to infer the number of dimensions from the length of `size`, but you may always specify it with `ndim`.

Note: The output will have `ndim+1` dimensions.

This is a generalization of `numpy.random.permutation()` to tensors. Otherwise it behaves the same.

multinomial(self, size=(), n=1, pvals=[0.5, 0.5], ndim=None):
Sample n times from a multinomial distribution defined by probabilities `pvals`, as many times as required by `size`. For instance, if `size=(p,q)`, `p*q` samples will be drawn, and the output shape will be `(p,q,len(pvals))`.

Theano tries to infer the number of dimensions from the length of `size`, but you may always specify it with `ndim`.

Note: The output will have `ndim+1` dimensions.

This is a generalization of `numpy.random.multinomial()` to the case where `n` and `pvals` are tensors. Otherwise it behaves the same.

shuffle_row_elements(self, input):
Return a variable with every row (rightmost index) shuffled.

This uses a permutation random variable internally, available via the `.permutation` attribute of the return value.

class `raw_random.`**RandomStateType**(*gof.Type*)
A *Type* for variables that will take `numpy.random.RandomState` values.

`raw_random.`**random_state_type**(*name=None*)
Return a new Variable whose `.type` is `random_state_type`.

class `raw_random.`**RandomFunction**(*gof.Op*)
Op that draws random numbers from a numpy.RandomState object. This Op is parametrized to draw numbers from many possible distributions.

`raw_random.`**uniform**(*random_state*, *size=None*, *low=0.0*, *high=1.0*, *ndim=None*, *dtype=None*)
Sample from a uniform distribution between low and high.

If the size argument is ambiguous on the number of dimensions, the first argument may be a plain integer to supplement the missing information.

Returns RandomVariable, NewRandomState

raw_random.**binomial**(*random_state*, *size=None*, *n=1*, *p=0.5*, *ndim=None*, *dtype='int64'*)
Sample n times with probability of success p for each trial and return the number of successes.

If size is ambiguous on the number of dimensions, ndim may be a plain integer to supplement the missing information.

Returns RandomVariable, NewRandomState

raw_random.**normal**(*random_state*, *size=None*, *avg=0.0*, *std=1.0*, *ndim=None*, *dtype=None*)
Sample from a normal distribution centered on avg with the specified standard deviation (std).

If size is ambiguous on the number of dimensions, ndim may be a plain integer to supplement the missing information.

Returns RandomVariable, NewRandomState

raw_random.**random_integers**(*random_state*, *size=None*, *low=0*, *high=1*, *ndim=None*, *dtype='int64'*)
Sample random integers in [low, high] to fill up size.

If size is ambiguous on the number of dimensions, ndim may be a plain integer to supplement the missing information.

Returns RandomVariable, NewRandomState

raw_random.**permutation**(*random_state*, *size=None*, *n=1*, *ndim=None*, *dtype='int64'*)
Returns permutations of the integers in [0, n[, as many times as required by size. For instance, if size=(p,q), p*q permutations will be generated, and the output shape will be (p,q,n), because each permutation is of size n.

If size is ambiguous on the number of dimensions, ndim may be a plain integer, which should correspond to len(size).

Note: The output will have ndim+1 dimensions.

Returns RandomVariable, NewRandomState

raw_random.**multinomial**(*random_state, size=None, p_vals=[0.5, 0.5], ndim=None, dtype='int64'*)
Sample from a multinomial distribution defined by probabilities pvals, as many times as required by size. For instance, if size=(p,q), p*q samples will be drawn, and the output shape will be (p,q,len(pvals)).

If size is ambiguous on the number of dimensions, ndim may be a plain integer, which should correspond to len(size).

Note: The output will have ndim+1 dimensions.

Returns `RandomVariable`, NewRandomState

`shared_randomstreams` – Friendly random numbers

Guide Since Theano uses a functional design, producing pseudo-random numbers in a graph is not quite as straightforward as it is in numpy.

The way to think about putting randomness into Theano's computations is to put random variables in your graph. Theano will allocate a numpy RandomState object for each such variable, and draw from it as necessary. We will call this sort of sequence of random numbers a *random stream*.

For an example of how to use random numbers, see *using_random_numbers*.

Reference

class `shared_randomstreams.`**`RandomStreams`**(*raw_random.RandomStreamsBase*)

This is a symbolic stand-in for `numpy.random.RandomState`. Random variables of various distributions are instantiated by calls to parent class `raw_random.RandomStreamsBase`.

`updates`()

Returns a list of all the (state, new_state) update pairs for the random variables created by this object

This can be a convenient shortcut to enumerating all the random variables in a large graph in the `update` parameter of function.

`seed`(*meta_seed*)

meta_seed will be used to seed a temporary random number generator, that will in turn generate seeds for all random variables created by this object (via *gen*).

Returns None

`gen`(*op*, **args*, ***kwargs*)

Return the random variable from *op(*args, **kwargs)*, but also install special attributes (`.rng` and `update`, see `RandomVariable`) into it.

This function also adds the returned variable to an internal list so that it can be seeded later by a call to *seed*.

`uniform, normal, binomial, multinomial, random_integers, ...`

See `raw_random.RandomStreamsBase`.

class `shared_randomstreams.`**`RandomVariable`**(*object*)

`rng`

The shared variable whose `.value` is the numpy RandomState generator feeding this random variable.

`update`

A pair whose first element is a shared variable whose value is a numpy RandomState, and whose second element is an [symbolic] expression for the next value of that RandomState after drawing

samples. Including this pair in the‘‘updates‘‘ list to function will cause the function to update the random number generator feeding this variable.

`signal` – Signal Processing

Signal Processing The signal subpackage contains ops which are useful for performing various forms of signal processing.

`conv` – Convolution

Note: Two similar implementation exists for conv2d:

`signal.conv2d` and `nnet.conv2d`.

The former implements a traditional 2D convolution, while the latter implements the convolutional layers present in convolutional neural networks (where filters are 3D and pool over several input channels).

`theano.tensor.signal.conv.`**`conv2d`**(*input*, *filters*, *image_shape=None*, *filter_shape=None*, *border_mode='valid'*, *subsample=(1, 1)*, ***kargs*)

signal.conv.conv2d performs a basic 2D convolution of the input with the given filters. The input parameter can be a single 2D image or a 3D tensor, containing a set of images. Similarly, filters can be a single 2D filter or a 3D tensor, corresponding to a set of 2D filters.

Shape parameters are optional and will result in faster execution.

Parameters

- **input** (*dmatrix of dtensor3*) – symbolic variable for images to be filtered
- **filters** (*dmatrix of dtensor3*) – symbolic variable containing filter values
- **border_mode** – ‘valid’ or ‘full’. see scipy.signal.convolve2d
- **subsample** – factor by which to subsample output
- **image_shape** (*tuple of length 2 or 3*) – ([number images,] image height, image width)
- **filter_shape** (*tuple of length 2 or 3*) – ([number filters,] filter height, filter width)
- **kwargs** – see theano.tensor.nnet.conv.conv2d

Return type symbolic 2D,3D or 4D tensor

Returns tensor of filtered images, with shape ([number images,] [number filters,] image height, image width)

`conv.`**`fft`**(**todo*)

[James has some code for this, but hasn’t gotten it into the source tree yet.]

`downsample` – Down-Sampling

theano.tensor.signal.downsample.**max_pool_2d**(*input*, *ds*, *ignore_border=False*, *st=None*, *padding=(0, 0)*, *mode='max'*)

Takes as input a N-D tensor, where N >= 2. It downscales the input image by the specified factor, by keeping only the maximum value of non-overlapping patches of size (ds[0],ds[1])

Parameters

- **input** (*N-D theano tensor of input images.*) – input images. Max pooling will be done over the 2 last dimensions.
- **ds** (*tuple of length 2*) – factor by which to downscale (vertical ds, horizontal ds). (2,2) will halve the image in each dimension.
- **ignore_border** (*bool*) – When True, (5,5) input with ds=(2,2) will generate a (2,2) output. (3,3) otherwise.
- **st** (*tuple of lenght 2*) – stride size, which is the number of shifts over rows/cols to get the the next pool region. if st is None, it is considered equal to ds (no overlap on pooling regions)
- **padding** (*tuple of two ints*) – (pad_h, pad_w), pad zeros to extend beyond four borders of the images, pad_h is the size of the top and bottom margins, and pad_w is the size of the left and right margins.
- **mode** (*string*) – 'max', 'sum', 'average_inc_pad' or 'average_exc_pad'. Operation executed on each window. *max* and *sum* always exclude the padding in the computation. *average* gives you the choice to include or exclude it.

theano.tensor.signal.downsample.**max_pool_2d_same_size**(*input*, *patch_size*)

Takes as input a 4-D tensor. It sets all non maximum values of non-overlapping patches of size (patch_size[0],patch_size[1]) to zero, keeping only the maximum values. The output has the same dimensions as the input.

Parameters

- **input** (*4-D theano tensor of input images.*) – input images. Max pooling will be done over the 2 last dimensions.
- **patch_size** (*tuple of length 2*) – size of the patch (patch height, patch width). (2,2) will retain only one non-zero value per patch of 4 values.

downsample.**fft**(**todo*)

[James has some code for this, but hasn't gotten it into the source tree yet.]

tensor.utils – Tensor Utils

theano.tensor.utils.**hash_from_ndarray**(*data*)

Return a hash from an ndarray

It takes care of the data, shapes, strides and dtype.

theano.tensor.utils.**shape_of_variables**(*fgraph*, *input_shapes*)

Compute the numeric shape of all intermediate variables given input shapes

Inputs: fgraph - the theano.FunctionGraph in question input_shapes - a dict mapping input to shape

Outputs: shapes - a dict mapping variable to shape

WARNING : This modifies the fgraph. Not pure.

```
>>> import theano
>>> x = theano.tensor.matrix('x')
>>> y = x[512:]; y.name = 'y'
>>> fgraph = theano.FunctionGraph([x], [y], clone=False)
>>> shape_of_variables(fgraph, {x: (1024, 1024)})
{y: (512, 1024), x: (1024, 1024)}
```

tensor.extra_ops – Tensor Extra Ops

class theano.tensor.extra_ops.**BinCountOp**(*minlength=None*)

DEPRECATED: use bincount() instead.

See function bincount for docstring

compatible_type = ('int8', 'int16', 'int32', 'int64', 'uint8', 'uint16', 'uint32', 'uint64')

Tuple of all compatible dtype for the parameter of this op.

class theano.tensor.extra_ops.**CpuContiguous**(*use_c_code='/usr/bin/g++'*)

Check to see if the input is c-contiguous, if it is, do nothing, else return a contiguous array

class theano.tensor.extra_ops.**Unique**(*return_index=False*, *return_inverse=False*, *return_counts=False*)

Wraps numpy.unique. This op is not implemented on the GPU.

```
>>> import numpy as np

>>> x = theano.tensor.vector()
>>> f = theano.function([x], Unique(True, True, False)(x))
>>> f([1, 2., 3, 4, 3, 2, 1.])
[array([ 1.,  2.,  3.,  4.]), array([0, 1, 2, 3]), array([0, 1, 2, 3, 2, 1, 0])]

>>> y = theano.tensor.matrix()
>>> g = theano.function([y], Unique(True, True, False)(y))
>>> g([[1, 1, 1.0], (2, 3, 3.0)])
[array([ 1.,  2.,  3.]), array([0, 3, 4]), array([0, 0, 0, 1, 2, 2])]
```

theano.tensor.extra_ops.**bartlett**(*M*)

An instance of this class returns the Bartlett spectral window in the time-domain. The Bartlett window is very similar to a triangular window, except that the end points are at zero. It is often used in signal processing for tapering a signal, without generating too much ripple in the frequency domain.

Parameters **M** – (integer scalar) Number of points in the output window. If zero or less, an empty vector is returned.

Returns (vector of doubles) The triangular window, with the maximum value normalized to one (the value one appears only if the number of samples is odd), with the first and last samples equal to zero.

New in version 0.6.

theano.tensor.extra_ops.**bincount**(*x*, *weights=None*, *minlength=None*, *assert_nonneg=False*)

Count number of occurrences of each value in array of ints.

The number of bins (of size 1) is one larger than the largest value in x. If minlength is specified, there will be at least this number of bins in the output array (though it will be longer if necessary, depending on the contents of x). Each bin gives the number of occurrences of its index value in x. If weights is specified the input array is weighted by it, i.e. if a value n is found at position i, out[n] += weight[i] instead of out[n] += 1.

Parameters

- **x** – 1 dimension, nonnegative ints
- **weights** – array of the same shape as x with corresponding weights. Optional.
- **minlength** – A minimum number of bins for the output array. Optional.
- **assert_nonneg** – A flag that inserts an assert_op to check if every input x is non-negative. Optional.

New in version 0.6.

theano.tensor.extra_ops.**compress**(*condition*, *x*, *axis=None*)

Return selected slices of an array along given axis.

It returns the input tensor, but with selected slices along a given axis retained. If no axis is provided, the tensor is flattened Corresponds to numpy.compress

Parameters

- **x** – Input data, tensor variable
- **condition** – 1 dimensional array of non-zero and zero values corresponding to indices of slices along a selected axis

Returns *x* with selected slices

New in version 0.7.

theano.tensor.extra_ops.**cumprod**(*x*, *axis=None*)

Return the cumulative product of the elements along a given axis.

Wraping of numpy.cumprod.

Parameters

- **x** – Input tensor variable.
- **axis** – The axis along which the cumulative product is computed. The default (None) is to compute the cumprod over the flattened array.

New in version 0.7.

theano.tensor.extra_ops.**cumsum**(*x*, *axis=None*)

Return the cumulative sum of the elements along a given axis.

Wraping of numpy.cumsum.

Parameters

- **x** – Input tensor variable.
- **axis** – The axis along which the cumulative sum is computed. The default (None) is to compute the cumsum over the flattened array.

New in version 0.7.

theano.tensor.extra_ops.**diff**(*x*, *n=1*, *axis=-1*)

Calculate the n-th order discrete difference along given axis.

The first order difference is given by out[i] = a[i + 1] - a[i] along the given axis, higher order differences are calculated by using diff recursively. Wraping of numpy.diff.

Parameters

- **x** – Input tensor variable.
- **n** – The number of times values are differenced, default is 1.
- **axis** – The axis along which the difference is taken, default is the last axis.

New in version 0.6.

theano.tensor.extra_ops.**fill_diagonal**(*a*, *val*)

Returns a copy of an array with all elements of the main diagonal set to a specified scalar value.

Parameters

- **a** – Rectangular array of at least two dimensions.
- **val** – Scalar value to fill the diagonal whose type must be compatible with that of array 'a' (i.e. 'val' cannot be viewed as an upcast of 'a').

Returns An array identical to 'a' except that its main diagonal is filled with scalar 'val'. (For an array 'a' with a.ndim >= 2, the main diagonal is the list of locations a[i, i, ..., i] (i.e. with indices all identical).)

Support rectangular matrix and tensor with more than 2 dimensions if the later have all dimensions are equals.

New in version 0.6.

theano.tensor.extra_ops.**fill_diagonal_offset**(*a*, *val*, *offset*)

Returns a copy of an array with all elements of the main diagonal set to a specified scalar value.

param a Rectangular array of two dimensions.

param val Scalar value to fill the diagonal whose type must be compatible with that of array 'a' (i.e. 'val' cannot be viewed as an upcast of 'a').

param offset Scalar value Offset of the diagonal from the main diagonal. Can be positive or negative integer.

return An array identical to 'a' except that its offset diagonal is filled with scalar 'val'. The output is unwrapped.

theano.tensor.extra_ops.**repeat**(*x*, *repeats*, *axis=None*)

Repeat elements of an array.

It returns an array which has the same shape as *x*, except along the given axis. The axis is used to speficy along which axis to repeat values. By default, use the flattened input array, and return a flat output array.

The number of repetitions for each element is *repeat*. *repeats* is broadcasted to fit the length of the given *axis*.

Parameters

- **x** – Input data, tensor variable.
- **repeats** – int, scalar or tensor variable.
- **axis** – int, optional.

See tensor.tile

New in version 0.6.

theano.tensor.extra_ops.**squeeze**(*x*)

Remove broadcastable dimensions from the shape of an array.

It returns the input array, but with the broadcastable dimensions removed. This is always *x* itself or a view into *x*.

Parameters **x** – Input data, tensor variable.

Returns *x* without its broadcastable dimensions.

New in version 0.6.

theano.tensor.extra_ops.**to_one_hot**(*y*, *nb_class*, *dtype=None*)

Return a matrix where each row correspond to the one hot encoding of each element in y.

param y A vector of integer value between 0 and nb_class - 1.

param nb_class The number of class in y.

param dtype The dtype of the returned matrix. Default floatX.

return A matrix of shape (y.shape[0], nb_class), where each row `i` is the one hot encoding of the corresponding `y[i]` value.

tensor.io – Tensor IO Ops

File operation

- Load from disk with the function `load` and its associated op `LoadFromDisk`

MPI operation

- Non-blocking transfer: `isend` and `irecv`.
- Blocking transfer: `send` and `recv`

Details

class theano.tensor.io.**LoadFromDisk**(*dtype*, *broadcastable*, *mmap_mode=None*)
An operation to load an array from disk

See Also load

@note: Non-differentiable.

class theano.tensor.io.**MPIRecv**(*source*, *tag*, *shape*, *dtype*)
An operation to asynchronously receive an array to a remote host using MPI

See Also MPIRecv MPIWait

@note: Non-differentiable.

class theano.tensor.io.**MPIRecvWait**(*tag*)
An operation to wait on a previously received array using MPI

See Also MPIRecv

@note: Non-differentiable.

class theano.tensor.io.**MPISend**(*dest*, *tag*)
An operation to asynchronously Send an array to a remote host using MPI

See Also MPIRecv MPISendWait

@note: Non-differentiable.

class theano.tensor.io.**MPISendWait**(*tag*)
An operation to wait on a previously sent array using MPI

See Also: MPISend

@note: Non-differentiable.

theano.tensor.io.**irecv**(*shape*, *dtype*, *source*, *tag*)
non-blocking receive

theano.tensor.io.**isend**(*var*, *dest*, *tag*)
Non blocking send

theano.tensor.io.**load**(*path*, *dtype*, *broadcastable*, *mmap_mode=None*)
Load an array from an .npy file.

Parameters

- **path** – A Generic symbolic variable, that will contain a string
- **dtype** – The data type of the array to be read.
- **broadcastable** – The broadcastable pattern of the loaded array, for instance, (False,) for a vector, (False, True) for a column, (False, False) for a matrix.
- **mmap_mode** – How the file will be loaded. None means that the data will be copied into an array in memory, 'c' means that the file will be mapped into virtual memory, so only the parts that are needed will be actually read from disk and put into memory. Other modes supported by numpy.load ('r', 'r+', 'w+') cannot be supported by Theano.

```
>>> from theano import *
>>> path = Variable(Generic())
>>> x = tensor.load(path, 'int64', (False,))
>>> y = x*2
>>> fn = function([path], y)
>>> fn("stored-array.npy")
array([0, 2, 4, 6, 8], dtype=int64)
```

theano.tensor.io.**mpi_send_wait_key**(*a*)
Wait as long as possible on Waits, Start Send/Recvs early

theano.tensor.io.**mpi_tag_key**(*a*)
Break MPI ties by using the variable tag - prefer lower tags first

theano.tensor.io.**recv**(*shape*, *dtype*, *source*, *tag*)
blocking receive

theano.tensor.io.**send**(*var*, *dest*, *tag*)
blocking send

tensor.slinalg – Linear Algebra Ops Using Scipy

Note: This module is not imported by default. You need to import it to use it.

API

class theano.tensor.slinalg.**Cholesky**(*lower=True*)
Return a triangular matrix square root of positive semi-definite *x*

L = cholesky(X, lower=True) implies dot(L, L.T) == X

class theano.tensor.slinalg.**CholeskyGrad**(*lower=True*)

perform(*node*, *inputs*, *outputs*)
Implements the "reverse-mode" gradient [3] for the Cholesky factorization of a positive-definite matrix.

class theano.tensor.slinalg.**Eigvalsh**(*lower=True*)
Generalized eigenvalues of a Hermetian positive definite eigensystem

class theano.tensor.slinalg.**EigvalshGrad**(*lower=True*)
Gradient of generalized eigenvalues of a Hermetian positive definite eigensystem

class theano.tensor.slinalg.**Expm**(*use_c_code='/usr/bin/g++'*)
Compute the matrix exponential of a square array

class theano.tensor.slinalg.**ExpmGrad**(*use_c_code='/usr/bin/g++'*)
Gradient of the matrix exponential of a square array.

[3] S. P. Smith. "Differentiation of the Cholesky Algorithm". Journal of Computational and Graphical Statistics, Vol. 4, No. 2 (Jun.,1995), pp. 134-147 http://www.jstor.org/stable/1390762

class theano.tensor.slinalg.**Solve**(*A_structure='general'*, *lower=False*, *overwrite_A=False*, *overwrite_b=False*)

Solve a system of linear equations

theano.tensor.slinalg.**kron**(*a*, *b*)

Kronecker product

Same as scipy.linalg.kron(a, b).

Note numpy.kron(a, b) != scipy.linalg.kron(a, b)! They don't have the same shape and order when a.ndim != b.ndim != 2.

Parameters

- **a** – array_like
- **b** – array_like

Returns array_like with a.ndim + b.ndim - 2 dimensions.

tensor.nlinalg – Linear Algebra Ops Using Numpy

Note: This module is not imported by default. You need to import it to use it.

API

class theano.tensor.nlinalg.**AllocDiag**(*use_c_code='/usr/bin/g++'*)

Allocates a square matrix with the given vector as its diagonal.

class theano.tensor.nlinalg.**Det**(*use_c_code='/usr/bin/g++'*)

Matrix determinant Input should be a square matrix

class theano.tensor.nlinalg.**Eig**(*use_c_code='/usr/bin/g++'*)

Compute the eigenvalues and right eigenvectors of a square array.

class theano.tensor.nlinalg.**Eigh**(*UPLO='L'*)

Return the eigenvalues and eigenvectors of a Hermitian or symmetric matrix.

grad(*inputs*, *g_outputs*)

The gradient function should return

$$\sum_n \left(W_n \frac{\partial w_n}{\partial a_{ij}} + \sum_k V_{nk} \frac{\partial v_{nk}}{\partial a_{ij}} \right),$$

where $[W, V]$ corresponds to `g_outputs`, a to `inputs`, and $(w, v) = \text{eig}(a)$.

Analytic formulae for eigensystem gradients are well-known in perturbation theory:

$$\frac{\partial\, w_n}{\partial a_{ij}} = v_{in}\, v_{jn}$$

$$\frac{\partial\, v_{kn}}{\partial a_{ij}} = \sum_{m \neq n} \frac{v_{km} v_{jn}}{w_n - w_m}$$

class theano.tensor.nlinalg.**EighGrad**(*UPLO='L'*)
Gradient of an eigensystem of a Hermitian matrix.

perform(*node*, *inputs*, *outputs*)
Implements the "reverse-mode" gradient for the eigensystem of a square matrix.

class theano.tensor.nlinalg.**ExtractDiag**(*view=False*)
Return the diagonal of a matrix.

Note work on the GPU.

perform(*node*, *ins*, *outs*)
For some reason numpy.diag(x) is really slow, so we implemented our own.

class theano.tensor.nlinalg.**MatrixInverse**
Computes the inverse of a matrix A.

Given a square matrix A, `matrix_inverse` returns a square matrix A_{inv} such that the dot product $A \cdot A_{inv}$ and $A_{inv} \cdot A$ equals the identity matrix I.

Note When possible, the call to this op will be optimized to the call of `solve`.

R_op(*inputs*, *eval_points*)
The gradient function should return

$$\frac{\partial X^{-1}}{\partial X} V,$$

where V corresponds to `g_outputs` and X to `inputs`. Using the matrix cookbook, once can deduce that the relation corresponds to

$$X^{-1} \cdot V \cdot X^{-1}.$$

grad(*inputs*, *g_outputs*)
The gradient function should return

$$V\frac{\partial X^{-1}}{\partial X},$$

where V corresponds to `g_outputs` and X to `inputs`. Using the matrix cookbook, once can deduce that the relation corresponds to

$$(X^{-1} \cdot V^T \cdot X^{-1})^T.$$

class `theano.tensor.nlinalg.`**`MatrixPinv`**

Computes the pseudo-inverse of a matrix A.

The pseudo-inverse of a matrix A, denoted A^+, is defined as: "the matrix that 'solves' [the least-squares problem] $Ax = b$," i.e., if $\bar{x}$ is said solution, then A^+ is that matrix such that $\bar{x} = A^+b$.

Note that $Ax = AA^+b$, so AA^+ is close to the identity matrix. This method is not faster then *matrix_inverse*. Its strength comes from that it works for non-square matrices. If you have a square matrix though, *matrix_inverse* can be both more exact and faster to compute. Also this op does not get optimized into a solve op.

class `theano.tensor.nlinalg.`**`QRFull`**(*mode*)

Full QR Decomposition. Computes the QR decomposition of a matrix. Factor the matrix a as qr, where q is orthonormal and r is upper-triangular.

class `theano.tensor.nlinalg.`**`QRIncomplete`**(*mode*)

Incomplete QR Decomposition. Computes the QR decomposition of a matrix. Factor the matrix a as qr and return a single matrix.

`theano.tensor.nlinalg.`**`diag`**(*x*)

Numpy-compatibility method If *x* is a matrix, return its diagonal. If *x* is a vector return a matrix with it as its diagonal.

•This method does not support the *k* argument that numpy supports.

`theano.tensor.nlinalg.`**`matrix_dot`**(**args*)

Shorthand for product between several dots

Given N matrices $A_0, A_1, .., A_N$, `matrix_dot` will generate the matrix product between all in the given order, namely $A_0 \cdot A_1 \cdot A_2 \cdot .. \cdot A_N$.

`theano.tensor.nlinalg.`**`qr`**(*a*, *mode='full'*)

Computes the QR decomposition of a matrix. Factor the matrix a as qr, where q is orthonormal and r is upper-triangular.

Parameters

- **a** (*array_like, shape (M, N)*) – Matrix to be factored.
- **mode** (*one of 'reduced', 'complete', 'r', 'raw', 'full' and 'economic', optional*) – If K = min(M, N), then

'reduced' returns q, r with dimensions (M, K), (K, N)

'complete' returns q, r with dimensions (M, M), (M, N)

'r' returns r only with dimensions (K, N)

'raw' returns h, tau with dimensions (N, M), (K,)

'full' alias of 'reduced', deprecated (default)

'economic' returns h from 'raw', deprecated.

The options 'reduced', 'complete', and 'raw' are new in numpy 1.8, see the notes for more information. The default is 'reduced' and to maintain backward compatibility with earlier versions of numpy both it and the old default 'full' can be omitted. Note that array h returned in 'raw' mode is transposed for calling Fortran. The 'economic' mode is deprecated. The modes 'full' and 'economic' may be passed using only the first letter for backwards compatibility, but all others must be spelled out.

Default mode is 'full' which is also default for numpy 1.6.1.

note Default mode was left to full as full and reduced are both doing the same thing in the new numpy version but only full works on the old previous numpy version.

Rtype q matrix of float or complex, optional

Return q A matrix with orthonormal columns. When mode = 'complete' the result is an orthogonal/unitary matrix depending on whether or not a is real/complex. The determinant may be either +/- 1 in that case.

Rtype r matrix of float or complex, optional

Return r The upper-triangular matrix.

`theano.tensor.nlinalg.`**`svd`**(*a*, *full_matrices=1*, *compute_uv=1*)

This function performs the SVD on CPU.

Parameters

- **full_matrices** (*bool, optional*) – If True (default), u and v have the shapes (M, M) and (N, N), respectively. Otherwise, the shapes are (M, K) and (K, N), respectively, where K = min(M, N).
- **compute_uv** (*bool, optional*) – Whether or not to compute u and v in addition to s. True by default.

Returns U, V and D matrices.

`theano.tensor.nlinalg.`**`trace`**(*X*)

Returns the sum of diagonal elements of matrix X.

Note work on GPU since 0.6rc4.

`gradient` – Symbolic Differentiation

Symbolic gradient is usually computed from `gradient.grad()`, which offers a more convenient syntax for the common case of wanting the gradient in some expressions with respect to a scalar cost. The `grad_sources_inputs()` function does the underlying work, and is more flexible, but is also more awkward to use when `gradient.grad()` can do the job. Driver for gradient calculations.

exception `theano.gradient.`**`DisconnectedInputError`**
Raised when grad is asked to compute the gradient with respect to a disconnected input and disconnected_inputs='raise'.

class `theano.gradient.`**`DisconnectedType`**
A type indicating that a variable is a result of taking the gradient of c with respect to x when c is not a function of x. A symbolic placeholder for 0, but to convey the extra information that this gradient is 0 because it is disconnected.

exception `theano.gradient.`**`GradientError`**(*arg*, *err_pos*, *abs_err*, *rel_err*, *abs_tol*, *rel_tol*)
This error is raised when a gradient is calculated, but incorrect.

`theano.gradient.`**`Lop`**(*f*, *wrt*, *eval_points*, *consider_constant=None*, *disconnected_inputs='raise'*)
Computes the L operation on *f* wrt to *wrt* evaluated at points given in *eval_points*. Mathematically this stands for the jacobian of *f* wrt to *wrt* left muliplied by the eval points.

Return type Variable or list/tuple of Variables depending on type of f

Returns symbolic expression such that L_op[i] = sum_i (d f[i] / d wrt[j]) eval_point[i] where the indices in that expression are magic multidimensional indices that specify both the position within a list and all coordinates of the tensor element in the last If *f* is a list/tuple, then return a list/tuple with the results.

exception `theano.gradient.`**`NullTypeGradError`**
Raised when grad encounters a NullType.

`theano.gradient.`**`Rop`**(*f*, *wrt*, *eval_points*)
Computes the R operation on *f* wrt to *wrt* evaluated at points given in *eval_points*. Mathematically this stands for the jacobian of *f* wrt to *wrt* right muliplied by the eval points.

Return type Variable or list/tuple of Variables depending on type of f

Returns symbolic expression such that R_op[i] = sum_j (d f[i] / d wrt[j]) eval_point[j] where the indices in that expression are magic multidimensional indices that specify both the position within a list and all coordinates of the tensor element in the last. If *wrt* is a list/tuple, then return a list/tuple with the results.

`theano.gradient.`**`consider_constant`**(*x*)
DEPRECATED: use zero_grad() or disconnected_grad() instead.

Consider an expression constant when computing gradients.

The expression itself is unaffected, but when its gradient is computed, or the gradient of another expression that this expression is a subexpression of, it will not be backpropagated through. In other words, the gradient of the expression is truncated to 0.

Parameters **x** – A Theano expression whose gradient should be truncated.

Returns The expression is returned unmodified, but its gradient is now truncated to 0.

New in version 0.7.

theano.gradient.**disconnected_grad**(*x*)

Consider an expression constant when computing gradients, while effectively not backpropagating through it.

The expression itself is unaffected, but when its gradient is computed, or the gradient of another expression that this expression is a subexpression of, it will not be backpropagated through. This is effectively equivalent to truncating the gradient expression to 0, but is executed faster than zero_grad(), which stilll has to go through the underlying computational graph related to the expression.

Parameters **x** – A Theano expression whose gradient should not be backpropagated through.

Returns The expression is returned unmodified, but its gradient is now effectively truncated to 0.

theano.gradient.**format_as**(*use_list*, *use_tuple*, *outputs*)

Formats the outputs according to the flags *use_list* and *use_tuple*. If *use_list* is True, *outputs* is returned as a list (if *outputs* is not a list or a tuple then it is converted in a one element list). If *use_tuple* is True, *outputs* is returned as a tuple (if *outputs* is not a list or a tuple then it is converted into a one element tuple). Otherwise (if both flags are false), *outputs* is returned.

theano.gradient.**grad**(*cost*, *wrt*, *consider_constant=None*, *disconnected_inputs='raise'*, *add_names=True*, *known_grads=None*, *return_disconnected='zero'*)

Return symbolic gradients for one or more variables with respect to some cost.

For more information about how automatic differentiation works in Theano, see `gradient`. For information on how to implement the gradient of a certain Op, see `grad()`.

Parameters

- **cost** (*Scalar (0-dimensional) tensor variable. May optionally be None if known_grads is provided.*) – a scalar with respect to which we are differentiating
- **wrt** (*Tensor variable or list of variables.*) – term[s] for which we want gradients
- **consider_constant** (*list of variables*) – a list of expressions not to backpropagate through
- **disconnected_inputs** (*string*) – Defines the behaviour if some of the variables in `wrt` are not part of the computational graph computing `cost` (or if all links are non-differentiable). The possible values are: - 'ignore': considers that the gradient on these parameters is zero. - 'warn': consider the gradient zero, and print a warning. - 'raise': raise DisconnectedInputError.
- **add_names** (*bool*) – If True, variables generated by grad will be named (d<cost.name>/d<wrt.name>) provided that both cost and wrt have names
- **known_grads** (*dict*) – If not None, a dictionary mapping variables to their gradients. This is useful in the case where you know the gradient on some variables

but do not know the original cost.

- **return_disconnected** (*string*) –
  - **'zero'** [If wrt[i] is disconnected, return value i will be] wrt[i].zeros_like()
  - **'None'** [If wrt[i] is disconnected, return value i will be] None
  - 'Disconnected' : returns variables of type DisconnectedType

Return type variable or list/tuple of Variables (matching *wrt*)

Returns symbolic expression of gradient of *cost* with respect to each of the *wrt* terms. If an element of *wrt* is not differentiable with respect to the output, then a zero variable is returned. It returns an object of same type as *wrt*: a list/tuple or Variable in all cases.

theano.gradient.**grad_clip**(*x*, *lower_bound*, *upper_bound*)

This op do a view in the forward, but clip the gradient.

This is an elemwise operation.

Parameters

- **x** – the variable we want its gradient inputs clipped
- **lower_bound** – The lower bound of the gradient value
- **upper_bound** – The upper bound of the gradient value.

Examples x = theano.tensor.scalar()

z = theano.tensor.grad(grad_clip(x, -1, 1)**2, x) z2 = theano.tensor.grad(x**2, x)

f = theano.function([x], outputs = [z, z2])

print(f(2.0)) # output (1.0, 4.0)

Note We register an opt in tensor/opt.py that remove the GradClip. So it have 0 cost in the forward and only do work in the grad.

theano.gradient.**grad_not_implemented**(*op*, *x_pos*, *x*, *comment=''*)

Return an un-computable symbolic variable of type *x.type*.

If any call to tensor.grad results in an expression containing this un-computable variable, an exception (NotImplementedError) will be raised indicating that the gradient on the *x_pos*'th input of *op* has not been implemented. Likewise if any call to theano.function involves this variable.

Optionally adds a comment to the exception explaining why this gradient is not implemented.

theano.gradient.**grad_undefined**(*op*, *x_pos*, *x*, *comment=''*)

Return an un-computable symbolic variable of type *x.type*.

If any call to tensor.grad results in an expression containing this un-computable variable, an exception (GradUndefinedError) will be raised indicating that the gradient on the *x_pos*'th input of *op* is mathematically undefined. Likewise if any call to theano.function involves this variable.

Optionally adds a comment to the exception explaining why this gradient is not defined.

theano.gradient.**hessian**(*cost*, *wrt*, *consider_constant=None*, *disconnected_inputs='raise'*)

Parameters

- **consider_constant** – a list of expressions not to backpropagate through
- **disconnected_inputs** (*string*) – Defines the behaviour if some of the variables in `wrt` are not part of the computational graph computing `cost` (or if all links are non-differentiable). The possible values are: - 'ignore': considers that the gradient on these parameters is zero. - 'warn': consider the gradient zero, and print a warning. - 'raise': raise an exception.

Returns either a instance of Variable or list/tuple of Variables (depending upon *wrt*) representing the Hessian of the *cost* with respect to (elements of) *wrt*. If an element of *wrt* is not differentiable with respect to the output, then a zero variable is returned. The return value is of same type as *wrt*: a list/tuple or TensorVariable in all cases.

`theano.gradient.`**`jacobian`**(*expression*, *wrt*, *consider_constant=None*, *disconnected_inputs='raise'*)

Parameters

- **consider_constant** – a list of expressions not to backpropagate through
- **disconnected_inputs** (*string*) – Defines the behaviour if some of the variables in `wrt` are not part of the computational graph computing `cost` (or if all links are non-differentiable). The possible values are: - 'ignore': considers that the gradient on these parameters is zero. - 'warn': consider the gradient zero, and print a warning. - 'raise': raise an exception.

Returns either a instance of Variable or list/tuple of Variables (depending upon *wrt*) repesenting the jacobian of *expression* with respect to (elements of) *wrt*. If an element of *wrt* is not differentiable with respect to the output, then a zero variable is returned. The return value is of same type as *wrt*: a list/tuple or TensorVariable in all cases.

class `theano.gradient.`**`numeric_grad`**(*f*, *pt*, *eps=None*, *out_type=None*)

Compute the numeric derivative of a scalar-valued function at a particular point.

static `abs_rel_err`(*a*, *b*)

Return absolute and relative error between a and b.

The relative error is a small number when a and b are close, relative to how big they are.

Formulas used: abs_err = abs(a - b) rel_err = abs_err / max(abs(a) + abs(b), 1e-8)

The denominator is clipped at 1e-8 to avoid dividing by 0 when a and b are both close to 0.

The tuple (abs_err, rel_err) is returned

`abs_rel_errors`(*g_pt*)

Return the abs and rel error of gradient estimate *g_pt*

g_pt must be a list of ndarrays of the same length as self.gf, otherwise a ValueError is raised.

Corresponding ndarrays in *g_pt* and *self.gf* must have the same shape or ValueError is raised.

`max_err`(*g_pt*, *abs_tol*, *rel_tol*)

Find the biggest error between g_pt and self.gf.

What is measured is the violation of relative and absolute errors, wrt the provided tolerances (abs_tol, rel_tol). A value > 1 means both tolerances are exceeded.

Return the argmax of min(abs_err / abs_tol, rel_err / rel_tol) over g_pt, as well as abs_err and rel_err at this point.

theano.gradient.**subgraph_grad**(*wrt*, *end*, *start=None*, *cost=None*, *details=False*)

With respect to *wrt*, computes gradients of cost and/or from existing *start* gradients, up to the *end* variables of a symbolic digraph. In other words, computes gradients for a subgraph of the symbolic theano function. Ignores all disconnected inputs.

This can be useful when one needs to perform the gradient descent iteratively (e.g. one layer at a time in an MLP), or when a particular operation is not differentiable in theano (e.g. stochastic sampling from a multinomial). In the latter case, the gradient of the non-differentiable process could be approximated by user-defined formula, which could be calculated using the gradients of a cost with respect to samples (0s and 1s). These gradients are obtained by performing a subgraph_grad from the *cost* or previously known gradients (*start*) up to the outputs of the stochastic process (*end*). A dictionary mapping gradients obtained from the user-defined differentiation of the process, to variables, could then be fed into another subgraph_grad as *start* with any other *cost* (e.g. weight decay).

In an MLP, we could use subgraph_grad to iteratively backpropagate:

```
x, t = theano.tensor.fvector('x'), theano.tensor.fvector('t')
w1 = theano.shared(np.random.randn(3,4))
w2 = theano.shared(np.random.randn(4,2))
a1 = theano.tensor.tanh(theano.tensor.dot(x,w1))
a2 = theano.tensor.tanh(theano.tensor.dot(a1,w2))
cost2 = theano.tensor.sqr(a2 - t).sum()
cost2 += theano.tensor.sqr(w2.sum())
cost1 = theano.tensor.sqr(w1.sum())

params = [[w2],[w1]]
costs = [cost2,cost1]
grad_ends = [[a1], [x]]

next_grad = None
param_grads = []
for i in xrange(2):
    param_grad, next_grad = theano.subgraph_grad(
        wrt=params[i], end=grad_ends[i],
        start=next_grad, cost=costs[i]
    )
    next_grad = dict(zip(grad_ends[i], next_grad))
    param_grads.extend(param_grad)
```

Parameters

- **wrt** (*list of variables*) – Gradients are computed with respect to *wrt*.
- **end** (*list of variables*) – Theano variables at which to end gradient descent (they are considered constant in theano.grad). For convenience, the gradients with respect to these variables are also returned.
- **start** (*dictionary of variables*) – If not None, a dictionary mapping variables to

their gradients. This is useful when the gradient on some variables are known. These are used to compute the gradients backwards up to the variables in *end* (they are used as known_grad in theano.grad).

- **cost** (*scalar (0-dimensional) variable*) – Additional costs for which to compute the gradients. For example, these could be weight decay, an l1 constraint, MSE, NLL, etc. May optionally be None if start is provided. Warning : If the gradients of *cost* with respect to any of the *start* variables is already part of the *start* dictionary, then it may be counted twice with respect to *wrt* and *end*.

  Warning: If the gradients of *cost* with respect to any of the *start* variables is already part of the *start* dictionary, then it may be counted twice with respect to *wrt* and *end*.

- **details** (*bool*) – When True, additionally returns the list of gradients from *start* and of *cost*, respectively, with respect to *wrt* (not *end*).

Return type Tuple of 2 or 4 Lists of Variables

Returns Returns lists of gradients with respect to *wrt* and *end*, respectively.

New in version 0.7.

`theano.gradient.`**`verify_grad`**(*fun*, *pt*, *n_tests=2*, *rng=None*, *eps=None*, *out_type=None*, *abs_tol=None*, *rel_tol=None*, *mode=None*, *cast_to_output_type=False*)

Test a gradient by Finite Difference Method. Raise error on failure.

Example:

```
>>> verify_grad(theano.tensor.tanh,
                (numpy.asarray([[2,3,4], [-1, 3.3, 9.9]]),),
                rng=numpy.random)
```

Raises an Exception if the difference between the analytic gradient and numerical gradient (computed through the Finite Difference Method) of a random projection of the fun's output to a scalar exceeds the given tolerance.

Parameters

- **fun** – a Python function that takes Theano variables as inputs, and returns a Theano variable. For instance, an Op instance with a single output.
- **pt** – the list of numpy.ndarrays to use as input values. These arrays must be either float32 or float64 arrays.
- **n_tests** – number of times to run the test
- **rng** – random number generator used to sample u, we test gradient of sum(u * fun) at pt
- **eps** – stepsize used in the Finite Difference Method (Default None is type-dependent) Raising the value of eps can raise or lower the absolute and relative

errors of the verification depending on the Op. Raising eps does not lower the verification quality for linear operations. It is better to raise eps than raising abs_tol or rel_tol.

- **out_type** – dtype of output, if complex (i.e. 'complex32' or 'complex64')
- **abs_tol** – absolute tolerance used as threshold for gradient comparison
- **rel_tol** – relative tolerance used as threshold for gradient comparison
- **cast_to_output_type** – if the output is float32 and cast_to_output_type is True, cast the random projection to float32. Otherwise it is float64.

Note WARNING to unit-test writers: if *op* is a function that builds a graph, try to make it a SMALL graph. Often verify grad is run in debug mode, which can be very slow if it has to verify a lot of intermediate computations.

Note This function does not support multiple outputs. In tests/test_scan.py there is an experimental verify_grad that covers that case as well by using random projections.

`theano.gradient.`**`zero_grad`**`(`*x*`)`

Consider an expression constant when computing gradients.

The expression itself is unaffected, but when its gradient is computed, or the gradient of another expression that this expression is a subexpression of, it will be backpropagated through with a value of zero. In other words, the gradient of the expression is truncated to 0.

Parameters **x** – A Theano expression whose gradient should be truncated.

Returns The expression is returned unmodified, but its gradient is now truncated to 0.

`config` – Theano Configuration

Guide

The config module contains many `attributes` that modify Theano's behavior. Many of these attributes are consulted during the import of the `theano` module and many are assumed to be read-only.

As a rule, the attributes in this module should not be modified by user code.

Theano's code comes with default values for these attributes, but you can override them from your .theanorc file, and override those values in turn by the `THEANO_FLAGS` environment variable.

The order of precedence is:

1. an assignment to theano.config.<property>
2. an assignment in `THEANO_FLAGS`
3. an assignment in the .theanorc file (or the file indicated in `THEANORC`)

You can print out the current/effective configuration at any time by printing theano.config. For example, to see a list of all active configuration variables, type this from the command-line:

```
python -c 'import theano; print theano.config' | less
```

Environment Variables

THEANO_FLAGS

This is a list of comma-delimited key=value pairs that control Theano's behavior.

For example, in bash, you can override your THEANORC defaults for <myscript>.py by typing this:

```
THEANO_FLAGS='floatX=float32,device=gpu0,nvcc.fastmath=True'  python <myscript>.py
```

If a value is defined several times in THEANO_FLAGS, the right-most definition is used. So, for instance, if THEANO_FLAGS='device=cpu,device=gpu0', then gpu0 will be used.

THEANORC

The location[s] of the .theanorc file[s] in ConfigParser format. It defaults to $HOME/.theanorc. On Windows, it defaults to $HOME/.theanorc:$HOME/.theanorc.txt to make Windows users' life easier.

Here is the .theanorc equivalent to the THEANO_FLAGS in the example above:

```
[global]
floatX = float32
device = gpu0

[nvcc]
fastmath = True
```

Configuration attributes that are available directly in config (e.g. config.device, config.mode) should be defined in the [global] section. Attributes from a subsection of config (e.g. config.nvcc.fastmath, config.blas.ldflags) should be defined in their corresponding section (e.g. [nvcc], [blas]).

Multiple configuration files can be specified by separating them with ':' characters (as in $PATH). Multiple configuration files will be merged, with later (right-most) files taking priority over earlier files in the case that multiple files specify values for a common configuration option. For example, to override system-wide settings with personal ones, set THEANORC=/etc/theanorc:~/.theanorc.

Config Attributes

The list below describes some of the more common and important flags that you might want to use. For the complete list (including documentation), import theano and print the config variable, as in:

```
python -c 'import theano; print theano.config' | less
```

config.**device**

String value: either 'cpu', 'gpu', 'gpu0', 'gpu1', 'gpu2', or 'gpu3'

Default device for computations. If gpu*, change the default to try to move computation to it and to put shared variable of float32 on it. Choose the default compute device for theano graphs. Setting

this to a gpu* string will make theano to try by default to move computation to it. Also it will make theano put by default shared variable of float32 on it. 'gpu' lets the driver select the GPU to use, while 'gpu?' makes Theano try to use a specific device. If we are not able to use the GPU, either we fall back on the CPU, or an error is raised, depending on the force_device flag.

This flag's value cannot be modified during the program execution.

Do not use upper case letters, only lower case even if NVIDIA use capital letters.

config.**force_device**
Bool value: either True or False

Default: False

If True and device=gpu*, we raise an error if we cannot use the specified device. If True and device=cpu, we disable the GPU. If False and device=gpu*, and if the specified device cannot be used, we warn and fall back to the CPU.

This is useful to run Theano's tests on a computer with a GPU, but without running the GPU tests.

This flag's value cannot be modified during the program execution.

config.**init_gpu_device**
String value: either '', 'gpu', 'gpu0', 'gpu1', 'gpu2', or 'gpu3'

Initialize the gpu device to use. When its value is gpu*, the theano flag device must be "cpu". Unlike device, setting this flag to a specific GPU will not try to use this device by default, in particular it will **not** move computations, nor shared variables, to the specified GPU.

This flag is useful to run GPU-specific tests on a particular GPU, instead of using the default one.

This flag's value cannot be modified during the program execution.

config.pycuda.**init**
Bool value: either True or False

Default: False

If True, always initialize PyCUDA when Theano want to initialize the GPU. With PyCUDA version 2011.2.2 or earlier, PyCUDA must initialize the GPU before Theano does it. Setting this flag to True, ensure that, but always import PyCUDA. It can be done manually by importing theano.misc.pycuda_init before Theano initialize the GPU device. Newer version of PyCUDA (currently only in the trunk) don't have this restriction.

config.**print_active_device**
Bool value: either True or False

Default: True

Print active device at when the GPU device is initialized.

config.**floatX**
String value: either 'float64' or 'float32'

Default: 'float64'

This sets the default dtype returned by tensor.matrix(), tensor.vector(), and similar functions. It also sets the default theano bit width for arguments passed as Python floating-point numbers.

config.**warn_float64**
String value: either 'ignore', 'warn', 'raise' or 'pdb'

Default: 'ignore'

When creating a TensorVariable with dtype float64, what should be done? This is useful to help find upcast to float64 in user code.

config.**allow_gc**
Bool value: either `True` or `False`

Default: `True`

This sets the default for the use of the Theano garbage collector for intermediate results. To use less memory, Theano frees the intermediate results as soon as they are no longer needed. Disabling Theano garbage collection allows Theano to reuse buffers for intermediate results between function calls. This speeds up Theano by no longer spending time reallocating space. This gives significant speed up on functions with many ops that are fast to execute, but this increases Theano's memory usage.

scan.**allow_output_prealloc**
Bool value, either `True` or `False`

Default: `True`

This enables, or not, an optimization in Scan in which it tries to pre-allocate memory for its outputs. Enabling the optimization can give a significant speed up with Scan at the cost of slightly increased memory usage.

config.**openmp**
Bool value: either True or False

Default: True if the environment variable OMP_NUM_THREADS!=1 or if we detect more than 1 CPU core. Otherwise False.

Enable or not parallel computation on the CPU with OpenMP. It is the default value used when creating an Op that support it. The best is to define it via Theano configuration file or with the environment variable THEANO_FLAGS.

config.**openmp_elemwise_minsize**
Positive int value, default: 200000.

This specifies the vectors minimum size for which elemwise ops use openmp, if openmp is enabled.

config.**cast_policy**
String value: either 'numpy+floatX' or 'custom'

Default: 'custom'

This specifies how data types are implicitly figured out in Theano, e.g. for constants or in the results of arithmetic operations. The 'custom' value corresponds to a set of custom rules originally used in Theano (which can be partially customized, see e.g. the in-code help of `tensor.NumpyAutocaster`), and will be deprecated in the future. The 'numpy+floatX' setting attempts to mimic the numpy casting rules, although it prefers to use float32 numbers instead of float64 when `config.floatX` is set to 'float32' and the user uses data that is not explicitly typed as float64 (e.g. regular Python floats). Note that 'numpy+floatX' is not currently behaving exactly as

planned (it is a work-in-progress), and thus you should consider it as experimental. At the moment it behaves differently from numpy in the following situations:

- Depending on the value of `config.int_division`, the resulting type of a division of integer types with the `/` operator may not match that of numpy.
- On mixed scalar / array operations, numpy tries to prevent the scalar from upcasting the array's type unless it is of a fundamentally different type. Theano does not attempt to do the same at this point, so you should be careful that scalars may upcast arrays when they would not when using numpy. This behavior should change in the near future.

`config.int_division`
String value: either 'int', 'floatX' or 'raise'

Default: 'int'

Specifies what to do when one tries to compute `x / y`, where both `x` and `y` are of integer types (possibly unsigned). 'int' means an integer is returned (as in Python 2.X), but this behavior is deprecated. 'floatX' returns a number of type given by `config.floatX`. 'raise' is the safest choice (and will become default in a future release of Theano) and raises an error when one tries to do such an operation, enforcing the use of the integer division operator (`//`) (if a float result is intended, either cast one of the arguments to a float, or use `x.__truediv__(y)`).

`config.mode`
String value: 'Mode', 'ProfileMode'(deprecated), 'DebugMode', 'FAST_RUN', 'FAST_COMPILE'

Default 'Mode'

This sets the default compilation mode for theano functions. By default the mode Mode is equivalent to FAST_RUN. See Config attribute linker and optimizer.

`config.profile`
Bool value: either True or False

Default False

Do the vm/cvm linkers profile the execution time of Theano functions?

See *Profiling Theano function* for examples.

`config.profile_memory`
Bool value: either True or False

Default False

Do the vm/cvm linkers profile the memory usage of Theano functions? It only works when profile=True.

`config.profile_optimizer`
Bool value: either True or False

Default False

Do the vm/cvm linkers profile the optimization phase when compiling a Theano function? It only works when profile=True.

config.profiling.**n_apply**
Positive int value, default: 20.

The number of Apply nodes to print in the profiler output

config.profiling.**n_ops**
Positive int value, default: 20.

The number of Ops to print in the profiler output

config.profiling.**min_memory_size**
Positive int value, default: 1024.

For the memory profile, do not print Apply nodes if the size of their outputs (in bytes) is lower than this.

config.profiling.**min_peak_memory**
Bool value: either True or False

Default False

Does the memory profile print the min peak memory usage? It only works when profile=True, profile_memory=True

config.profiling.**destination**
String value: 'stderr', 'stdout', or a name of a file to be created

Default 'stderr'

Name of the destination file for the profiling output. The profiling output can be either directed to stderr (default), or stdout or an arbitrary file.

config.lib.**amdlibm**
Bool value: either True or False

Default False

This makes the compilation use the amdlibm library, which is faster than the standard libm.

config.lib.**cnmem**
Float value: >= 0

Do we enable CNMeM or not (a faster CUDA memory allocator). In Theano dev version until 0.7.1 is released.

That library is included in Theano, you do not need to install it.

The value represents the start size (in MB or % of total GPU memory) of the memory pool. If more memory are needed, it will try to get more, but this can cause more memory fragmentation:

- 0: not enabled.
- 0 < N <= 1: % of the total GPU memory (clipped to .985 for driver memory)
- > 0: use that number of MB of memory.

Default 0 (but should change later)

Note: This could cause memory fragmentation. So if you have a memory error while using cnmem,

try to allocate more memory at the start or disable it. If you try this, report your result on :ref‘theano-dev‘.

---

config.**linker**
String value: ‘c|py’, ‘py’, ‘c’, ‘c|py_nogc’

Default: ‘c|py’

When the mode is Mode, it sets the default linker used. See *Configuration Settings and Compiling Modes* for a comparison of the different linkers.

config.**optimizer**
String value: ‘fast_run’, ‘merge’, ‘fast_compile’, ‘None’

Default: ‘fast_run’

When the mode is Mode, it sets the default optimizer used.

config.**on_opt_error**

String value: ‘warn’, ‘raise’, ‘pdb’ or ‘ignore’

Default: ‘warn’

When a crash occurs while trying to apply some optimization, either warn the user and skip this optimization (‘warn’), raise the exception (‘raise’), fall into the pdb debugger (‘pdb’) or ignore it (‘ignore’). We suggest to never use ‘ignore’ except in tests.

If you encounter a warning, report it on theano-dev.

config.**assert_no_cpu_op**
String value: ‘ignore’ or ‘warn’ or ‘raise’ or ‘pdb’

Default: ‘ignore’

If there is a CPU op in the computational graph, depending on its value; this flag can either raise a warning, an exception or stop the compilation with pdb.

config.**on_shape_error**
String value: ‘warn’ or ‘raise’

Default: ‘warn’

When an exception is raised when inferring the shape of some apply node, either warn the user and use a default value (‘warn’), or raise the exception (‘raise’).

config.warn.**ignore_bug_before**
String value: ‘None’, ‘all’, ‘0.3’, ‘0.4’, ‘0.4.1’, ‘0.5’, ‘0.6’, ‘0.7’

Default: ‘0.6’

When we fix a Theano bug that generated bad results under some circumstances, we also make Theano raise a warning when it encounters the same circumstances again. This helps to detect if said bug had affected your past experiments, as you only need to run your experiment again with the new version, and you do not have to understand the Theano internal that triggered the bug. A better way to detect this will be implemented. See this ticket.

This flag allows new users not to get warnings about old bugs, that were fixed before their first checkout of Theano. You can set its value to the first version of Theano that you used (probably 0.3 or higher)

None means that all warnings will be displayed. *all* means all warnings will be ignored.

It is recommended that you put a version, so that you will see future warnings. It is also recommended you put this into your .theanorc, so this setting will always be used.

This flag's value cannot be modified during the program execution.

`config.`**`base_compiledir`**
Default: On Windows: $LOCALAPPDATA\Theano if $LOCALAPPDATA is defined, otherwise and on other systems: ~/.theano.

This directory stores the platform-dependent compilation directories.

This flag's value cannot be modified during the program execution.

`config.`**`compiledir_format`**
Default: "compiledir_%(platform)s-%(processor)s-%(python_version)s-%(python_bitwidth)s"

This is a Python format string that specifies the subdirectory of `config.base_compiledir` in which to store platform-dependent compiled modules. To see a list of all available substitution keys, run `python -c "import theano; print theano.config"`, and look for compiledir_format.

This flag's value cannot be modified during the program execution.

`config.`**`compiledir`**
Default: `config.base_compiledir/config.compiledir_format`

This directory stores dynamically-compiled modules for a particular platform.

This flag's value cannot be modified during the program execution.

`config.blas.`**`ldflags`**
Default: '-lblas'

Link arguments to link against a (Fortran) level-3 blas implementation. The default will test if '-lblas' work. If not, we will disable our c code for BLAS.

`config.experimental.`**`local_alloc_elemwise_assert`**
Bool value: either True or False

Default: True

When the local_alloc_optimization is applied, add an assert to highlight shape errors.

Without such asserts this optimization could hide errors in the user code. We add the assert only if we can't infer that the shapes are equivalent. As such this optimization does not always introduce an assert in the graph. Removing the assert could speed up execution.

`config.cuda.`**`root`**
Default: $CUDA_ROOT or failing that, "/usr/local/cuda"

A directory with bin/, lib/, include/ folders containing cuda utilities.

config.dnn.conv.**workmem**
String value: 'none', 'small', 'large'

Default: 'small'

The default value for the amount of working memory that is tolerated in the convolution implementation in cudnn.

'none' Don't allow any extra memory.

'small' Allow extra memory that is much smaller than the input sizes.

'large' Allow extra memory that is on the order of the input sizes.

config.gcc.**cxxflags**
Default: ""

Extra parameters to pass to gcc when compiling. Extra include paths, library paths, configuration options, etc.

config.**cxx**
Default: Full path to g++ if g++ is present. Empty string otherwise.

Indicates which C++ compiler to use. If empty, no C++ code is compiled. Theano automatically detects whether g++ is present and disables C++ compilation when it is not. On darwin systems (Mac OS X), it preferably looks for clang++ and uses that if available.

We print a warning if we detect that no compiler is present. It is recommended to run with C++ compilation as Theano will be much slower otherwise.

This can be any compiler binary (full path or not) but things may break if the interface is not g++-compatible to some degree.

config.nvcc.**fastmath**
Default: False

If true, this will enable fastmath (–use_fast_math) mode for compiled cuda code which makes div and sqrt faster at the cost of precision. This also disables support for denormal numbers.

config.**optimizer_excluding**
Default: ""

A list of optimizer tags that we don't want included in the default Mode. If multiple tags, separate them by ':'. Ex: to remove the elemwise inplace optimizer(slow for big graph), use the flags: optimizer_excluding:inplace_opt, where inplace_opt is the name of that optimization.

This flag's value cannot be modified during the program execution.

config.**optimizer_including**
Default: ""

A list of optimizer tags that we want included in the default Mode. If multiple tags, separate them by ':'.

This flag's value cannot be modified during the program execution.

config.**optimizer_requiring**
Default: ""

A list of optimizer tags that we require for optimizer in the default Mode. If multiple tags, separate them by ':'.

This flag's value cannot be modified during the program execution.

config.**optimizer_verbose**
Bool value: either True or False

Default: False

When True, we print on the stdout the optimization applied.

config.**nocleanup**
Bool value: either True or False

Default: False

If False, source code files are removed when they are not needed anymore. This means files whose compilation failed are deleted. Set to True to keep those files in order to debug compilation errors.

config.**compile**
This section contains attributes which influence the compilation of C code for ops. Due to historical reasons many attributes outside of this section also have an influence over compilation, most notably 'cxx'. This is not expected to change any time soon.

config.compile.**timeout**
Positive int value, default: `compile.wait` * 24

Time to wait before an unrefreshed lock is broken and stolen. This is in place to avoid manual cleanup of locks in case a process crashed and left a lock in place.

The refresh time is automatically set to half the timeout value.

config.compile.**wait**
Positive int value, default: 5

Time to wait between attempts at grabbing the lock if the first attempt is not successful. The actual time will be between `compile.wait` and `compile.wait` * 2 to avoid a crowding effect on lock.

config.**DebugMode**
This section contains various attributes configuring the behaviour of mode `DebugMode`. See directly this section for the documentation of more configuration options.

config.DebugMode.**check_preallocated_output**
Default: `''`

A list of kinds of preallocated memory to use as output buffers for each Op's computations, separated by `:`. Implemented modes are:

- `"initial"`: initial storage present in storage map (for instance, it can happen in the inner function of Scan),
- `"previous"`: reuse previously-returned memory,
- `"c_contiguous"`: newly-allocated C-contiguous memory,
- `"f_contiguous"`: newly-allocated Fortran-contiguous memory,

- "strided": non-contiguous memory with various stride patterns,
- "wrong_size": memory with bigger or smaller dimensions,
- "ALL": placeholder for all of the above.

In order not to test with preallocated memory, use an empty string, "".

config.DebugMode.**check_preallocated_output_ndim**
Positive int value, default: 4.

When testing with "strided" preallocated output memory, test all combinations of strides over that number of (inner-most) dimensions. You may want to reduce that number to reduce memory or time usage, but it is advised to keep a minimum of 2.

config.DebugMode.**warn_input_not_reused**
Bool value, default: True

Generate a warning when the destroy_map or view_map tell that an op work inplace, but the op did not reuse the input for its output.

config.**numpy**
This section contains different attributes for configuring numpy's behaviour, described by numpy.seterr.

config.numpy.**seterr_all**
String Value: 'ignore','warn','raise','call','print','log','None'

Default: 'ignore'

Set the default behaviour described by numpy.seterr.

'None' means that numpy's default behaviour will not be changed (unless one of the other *config.numpy.seterr_** overrides it), but this behaviour can change between numpy releases.

This flag sets the default behaviour for all kinds of floating-pont errors, and it can be overriden for specific errors by setting one (or more) of the flags below.

This flag's value cannot be modified during the program execution.

config.numpy.**seterr_divide**
String Value: 'None','ignore','warn','raise','call','print','log'

Default: 'None'

Sets numpy's behavior for division by zero. 'None' means using the default, defined by config.numpy.seterr_all.

This flag's value cannot be modified during the program execution.

config.numpy.**seterr_over**
String Value: 'None','ignore','warn','raise','call','print','log'

Default: 'None'

Sets numpy's behavior for floating-point overflow. 'None' means using the default, defined by config.numpy.seterr_all.

This flag's value cannot be modified during the program execution.

config.numpy.**seterr_under**
String Value: 'None','ignore','warn','raise','call','print','log'

Default: 'None'

Sets numpy's behavior for floating-point underflow. 'None' means using the default, defined by config.numpy.seterr_all.

This flag's value cannot be modified during the program execution.

config.numpy.**seterr_invalid**
String Value: 'None','ignore','warn','raise','call','print','log'

Default: 'None'

Sets numpy's behavior for invalid floating-point operation. 'None' means using the default, defined by config.numpy.seterr_all.

This flag's value cannot be modified during the program execution.

config.**compute_test_value**
String Value: 'off','ignore','warn','raise'.

Default: 'off'

Setting this attribute to something other than 'off' activates a debugging mechanism, where Theano executes the graph on-the-fly, as it is being built. This allows the user to spot errors early on (such as dimension mis-match), **before** optimizations are applied.

Theano will execute the graph using the Constants and/or shared variables provided by the user. Purely symbolic variables (e.g. x = T.dmatrix()) can be augmented with test values, by writing to their 'tag.test_value' attribute (e.g. x.tag.test_value = numpy.random.rand(5,4)).

When not 'off', the value of this option dictates what happens when an Op's inputs do not provide appropriate test values:

- 'ignore' will silently skip the debug mechanism for this Op
- 'warn' will raise a UserWarning and skip the debug mechanism for this Op
- 'raise' will raise an Exception

config.**compute_test_value_opt**
As compute_test_value, but it is the value used during Theano optimization phase. Theano user's do not need to use this. This is to help debug shape error in Theano optimization.

config.**reoptimize_unpickled_function**
Bool value, default: True

Theano users can use the standard python pickle tools to save a compiled theano function. When pickling, both graph before and after the optimization are saved, including shared variables. When set to True, the graph is reoptimized when being unpickled. Otherwise, skip the graph optimization and use directly the optimized graph.

config.**exception_verbosity**
String Value: 'low','high'.

Default: 'low'

If 'low', the text of exceptions will generally refer to apply nodes with short names such as 'Elemwise{add_no_inplace}'. If 'high', some exceptions will also refer to apply nodes with long descriptions like:

```
A. Elemwise{add_no_inplace}
      B. log_likelihood_v_given_h
      C. log_likelihood_h
```

config.cmodule.**warn_no_version**
Bool value, default: False

If True, will print a warning when compiling one or more Op with C code that can't be cached because there is no c_code_cache_version() function associated to at least one of those Ops.

config.cmodule.**mac_framework_link**
Bool value, default: False

If set to True, breaks certain MacOS installations with the infamous Bus Error.

config.cmodule.**remove_gxx_opt**
Bool value, default: False

If True, will remove the -O* parameter passed to g++. This is useful to debug in gdb modules compiled by Theano. The parameter -g is passed by default to g++.

config.cmodule.**compilation_warning**
Bool value, default: False

If True, will print compilation warnings.

config.cmodule.**preload_cache**
Bool value, default: False

If set to True, will preload the C module cache at import time

printing – Graph Printing and Symbolic Print Statement

Guide

Printing during execution Intermediate values in a computation cannot be printed in the normal python way with the print statement, because Theano has no *statements*. Instead there is the Print Op.

```
>>> x = T.dvector()
>>> hello_world_op = printing.Print('hello world')
>>> printed_x = hello_world_op(x)
>>> f = function([x], printed_x)
>>> f([1, 2, 3])
>>> # output: "hello world __str__ = [ 1.  2.  3.]"
```

If you print more than one thing in a function like *f*, they will not necessarily be printed in the order that you think. The order might even depend on which graph optimizations are applied. Strictly speaking, the order of printing is not completely defined by the interface – the only hard rule is that if the input of some print

output *a* is ultimately used as an input to some other print input *b* (so that *b* depends on *a*), then *a* will print before *b*.

Printing graphs Theano provides two functions (`theano.pp()` and `theano.printing.debugprint()`) to print a graph to the terminal before or after compilation. These two functions print expression graphs in different ways: `pp()` is more compact and math-like, `debugprint()` is more verbose. Theano also provides `theano.printing.pydotprint()` that creates a png image of the function.

1. The first is `theano.pp()`.

```
>>> x = T.dscalar('x')
>>> y = x ** 2
>>> gy = T.grad(y, x)
>>> pp(gy)  # print out the gradient prior to optimization
'((fill((x ** 2), 1.0) * 2) * (x ** (2 - 1)))'
>>> f = function([x], gy)
>>> pp(f.maker.fgraph.outputs[0])
'(2.0 * x)'
```

The parameter in T.dscalar('x') in the first line is the name of this variable in the graph. This name is used when printing the graph to make it more readable. If no name is provided the variable x is printed as its type as returned by x.type(). In this example - <TensorType(float64, scalar)>.

The name parameter can be any string. There are no naming restrictions: in particular, you can have many variables with the same name. As a convention, we generally give variables a string name that is similar to the name of the variable in local scope, but you might want to break this convention to include an object instance, or an iteration number or other kinds of information in the name.

Note: To make graphs legible, `pp()` hides some Ops that are actually in the graph. For example, automatic DimShuffles are not shown.

---

2. The second function to print a graph is `theano.printing.debugprint()`

```
>>> theano.printing.debugprint(f.maker.fgraph.outputs[0])
Elemwise{mul,no_inplace} [@A] ''
 |TensorConstant{2.0} [@B]
 |x [@C]
```

Each line printed represents a Variable in the graph. The line `|x [@C` means the variable named `x` with debugprint identifier [@C] is an input of the Elemwise. If you accidentally have two variables called `x` in your graph, their different debugprint identifier will be your clue.

The line `|TensorConstant{2.0} [@B]` means that there is a constant 2.0 with this debugprint identifier.

The line `Elemwise{mul,no_inplace} [@A] ''` is indented less than the other ones, because it means there is a variable computed by multiplying the other (more indented) ones together.

The `|` symbol are just there to help read big graph. The group together inputs to a node.

Sometimes, you'll see a Variable but not the inputs underneath. That can happen when that Variable has already been printed. Where else has it been printed? Look for debugprint identifier using the Find feature of your text editor.

```
>>> theano.printing.debugprint(gy)
Elemwise{mul} [@A] ''
 |Elemwise{mul} [@B] ''
 | |Elemwise{second,no_inplace} [@C] ''
 | | |Elemwise{pow,no_inplace} [@D] ''
 | | | |x [@E]
 | | | |TensorConstant{2} [@F]
 | | |TensorConstant{1.0} [@G]
 | |TensorConstant{2} [@F]
 |Elemwise{pow} [@H] ''
   |x [@E]
   |Elemwise{sub} [@I] ''
     |TensorConstant{2} [@F]
     |InplaceDimShuffle{} [@J] ''
       |TensorConstant{1} [@K]

>>> theano.printing.debugprint(gy, depth=2)
Elemwise{mul} [@A] ''
 |Elemwise{mul} [@B] ''
 |Elemwise{pow} [@C] ''
```

If the depth parameter is provided, it limits the number of levels that are shown.

3. The function `theano.printing.pydotprint()` will print a compiled theano function to a png file.

In the image, Apply nodes (the applications of ops) are shown as ellipses and variables are shown as boxes. The number at the end of each label indicates graph position. Boxes and ovals have their own set of positions, so you can have apply #1 and also a variable #1. The numbers in the boxes (Apply nodes) are actually their position in the run-time execution order of the graph. Green ovals are inputs to the graph and blue ovals are outputs.

If your graph uses shared variables, those shared variables will appear as inputs. Future versions of the `pydotprint()` may distinguish these inplicit inputs from explicit inputs.

If you give updates arguments when creating your function, these are added as extra inputs and outputs to the graph. Future versions of `pydotprint()` may distinguish these implicit inputs and outputs from explicit inputs and outputs.

Reference

class `printing.`**`Print`**(*Op*)

This identity-like Op has the side effect of printing a message followed by its inputs when it runs. Default behaviour is to print the __str__ representation. Optionally, one can pass a list of the input member functions to execute, or attributes to print.

`__init__`(*message=""*, *attrs=("__str__")*

Parameters

- **message** (*string*) – prepend this to the output
- **attrs** (*list of strings*) – list of input node attributes or member functions to print. Functions are identified through callable(), executed and their return value printed.

__call__(*x*)

Parameters **x** (a `Variable`) – any symbolic variable

Returns symbolic identity(x)

When you use the return-value from this function in a theano function, running the function will print the value that *x* takes in the graph.

`theano.printing.`**debugprint**(*obj*, *depth=-1*, *print_type=False*, *file=None*, *ids='CHAR'*, *stop_on_name=False*, *done=None*)

Print a computation graph as text to stdout or a file.

Parameters

- **obj** (*Variable, Apply, or Function instance*) – symbolic thing to print
- **depth** (*integer*) – print graph to this depth (-1 for unlimited)
- **print_type** (*boolean*) – whether to print the type of printed objects
- **file** (*None, 'str', or file-like object*) – print to this file ('str' means to return a string)
- **ids** (*str*) – How do we print the identifier of the variable id - print the python id value int - print integer character CHAR - print capital character "" - don't print an identifier
- **stop_on_name** – When True, if a node in the graph has a name, we don't print anything below it.
- **done** (*None or dict*) – A dict where we store the ids of printed node. Useful to have multiple call to debugprint share the same ids.

Returns string if *file* == 'str', else file arg

Each line printed represents a Variable in the graph. The indentation of lines corresponds to its depth in the symbolic graph. The first part of the text identifies whether it is an input (if a name or type is printed) or the output of some Apply (in which case the Op is printed). The second part of the text is an identifier of the Variable. If print_type is True, we add a part containing the type of the Variable

If a Variable is encountered multiple times in the depth-first search, it is only printed recursively the first time. Later, just the Variable identifier is printed.

If an Apply has multiple outputs, then a '.N' suffix will be appended to the Apply's identifier, to indicate which output a line corresponds to.

`theano.`**pp**(**args*)

Just a shortcut to `theano.printing.pp()`

`theano.printing.`**pp**(**args*)

theano.printing.**pydotprint**(*fct*, *outfile=None*, *compact=True*, *format='png'*, *with_ids=False*, *high_contrast=True*, *cond_highlight=None*, *colorCodes=None*, *max_label_size=70*, *scan_graphs=False*, *var_with_name_simple=False*, *print_output_file=True*, *return_image=False*)

Print to a file the graph of a compiled theano function's ops. Supports all pydot output formats, including png and svg.

Parameters

- **fct** – a compiled Theano function, a Variable, an Apply or a list of Variable.
- **outfile** – the output file where to put the graph.
- **compact** – if True, will remove intermediate var that don't have name.
- **format** – the file format of the output.
- **with_ids** – Print the toposort index of the node in the node name. and an index number in the variable ellipse.
- **high_contrast** – if true, the color that describes the respective node is filled with its corresponding color, instead of coloring the border
- **colorCodes** – dictionary with names of ops as keys and colors as values
- **cond_highlight** – Highlights a lazy if by sorrounding each of the 3 possible categories of ops with a border. The categories are: ops that are on the left branch, ops that are on the right branch, ops that are on both branches As an alternative you can provide the node that represents the lazy if
- **scan_graphs** – if true it will plot the inner graph of each scan op in files with the same name as the name given for the main file to which the name of the scan op is concatenated and the index in the toposort of the scan. This index can be printed with the option with_ids.
- **var_with_name_simple** – If true and a variable have a name, we will print only the variable name. Otherwise, we concatenate the type to the var name.
- **return_image** – If True, it will create the image and return it. Useful to display the image in ipython notebook.

```
import theano
v = theano.tensor.vector()
from IPython.display import SVG
SVG(theano.printing.pydotprint(v*2, return_image=True,
                               format='svg'))
```

In the graph, ellipses are Apply Nodes (the execution of an op) and boxes are variables. If variables have names they are used as text (if multiple vars have the same name, they will be merged in the graph). Otherwise, if the variable is constant, we print its value and finally we print the type + a unique number to prevent multiple vars from being merged. We print the op of the apply in the Apply

box with a number that represents the toposort order of application of those Apply. If an Apply has more than 1 input, we label each edge between an input and the Apply node with the input's index.

Variable color code::

- Cyan boxes are SharedVariable, inputs and/or outputs) of the graph,
- Green boxes are inputs variables to the graph,
- Blue boxes are outputs variables of the graph,
- Grey boxes are variables that are not outputs and are not used,

Default apply node code::

- Red ellipses are transfers from/to the gpu
- Yellow are scan node
- Brown are shape node
- Magenta are IfElse node
- Dark pink are elemwise node
- Purple are subtensor
- Orange are alloc node

For edges, they are black by default. If a node returns a view of an input, we put the corresponding input edge in blue. If it returns a destroyed input, we put the corresponding edge in red.

Note: Since October 20th, 2014, this print the inner function of all scan separately after the top level debugprint output.

`compile` – Transforming Expression Graphs to Functions

`shared` - defines theano.shared

class `shared.`**`SharedVariable`**

Variable with Storage that is shared between functions that it appears in. These variables are meant to be created by registered *shared constructors* (see `shared_constructor()`).

The user-friendly constructor is `shared()`

`value`

Read/write access to the [non-symbolic] value/data associated with this SharedVariable.

Changes to this value will be visible to all functions using this SharedVariable.

`__init__`(*self*, *name*, *type*, *value*, *strict*, *container=None*)

Parameters

- **name** (*None or str*) – The name for this variable.

- **type** – The *Type* for this Variable.
- **value** – A value to associate with this variable (a new container will be created).
- **strict** – True -> assignments to `self.value` will not be casted or copied, so they must have the correct type or an exception will be raised.
- **container** – The container to use for this variable. This should instead of the *value* parameter. Using both is an error.

container
A container to use for this SharedVariable when it is an implicit function parameter.

Type class:*Container*

`theano.compile.sharedvalue.`**shared**(*value*, *name=None*, *strict=False*, *allow_downcast=None*, ***kwargs*)
Return a SharedVariable Variable, initialized with a copy or reference of *value*.

This function iterates over *constructor functions* to find a suitable SharedVariable subclass. The suitable one is the first constructor that accept the given value.

This function is meant as a convenient default. If you want to use a specific shared variable constructor, consider calling it directly.

`theano.shared` is a shortcut to this function.

By passing kwargs, you effectively limit the set of potential constructors to those that can accept those kwargs.

Some shared variable have `borrow` as extra kwargs. See for details.

Some shared variable have `broadcastable` as extra kwargs. As shared variable shapes can change, all dimensions default to not being broadcastable, even if `value` has a shape of 1 along some dimension. This parameter allows you to create for example a *row* or *column* 2d tensor.

`shared.`**constructors**

A list of shared variable constructors that will be tried in reverse order.

`shared.`**shared_constructor**(*ctor*)
Append *ctor* to the list of shared constructors (see `shared()`).

Each registered constructor `ctor` will be called like this:

```
ctor(value, name=name, strict=strict, **kwargs)
```

If it do not support given value, it must raise a TypeError.

`function` - defines theano.function

Guide This module provides `function()`, commonly accessed as *theano.function*, the interface for compiling graphs into callable objects.

You've already seen example usage in the basic tutorial... something like this:

```
>>> x = theano.tensor.dscalar()
>>> f = theano.function([x], 2*x)
>>> print f(4)                      # prints 8.0
```

The idea here is that we've compiled the symbolic graph (`2*x`) into a function that can be called on a number and will do some computations.

The behaviour of function can be controlled in several ways, such as `Param`, `mode`, `updates`, and `givens`. These are covered in the *tutorial examples* and *tutorial on modes*.

Reference

class `function.`**Out**

A class for attaching information to function outputs

variable

A variable in an expression graph to use as a compiled-function output

borrow

`True` indicates that a reference to internal storage may be returned, and that the caller is aware that subsequent function evaluations might overwrite this memory.

__init__(*variable*, *borrow=False*)

Initialize attributes from arguments.

class `function.`**Param**

A class for attaching information to function inputs.

variable

A variable in an expression graph to use as a compiled-function parameter

default

The default value to use at call-time (can also be a Container where the function will find a value at call-time.)

name

A string to identify an argument for this parameter in keyword arguments.

mutable

`True` means the compiled-function is allowed to modify this argument. `False` means it is not allowed.

strict

If `False`, a function argument may be copied or cast to match the type required by the parameter *variable*. If `True`, a function argument must exactly match the type required by *variable*.

__init__(*self*, *variable*, *default=None*, *name=None*, *mutable=False*, *strict=False*)

Initialize object attributes.

`function.`**function**(*inputs*, *outputs*, *mode=None*, *updates=None*, *givens=None*, *no_default_updates=False*, *accept_inplace=False*, *name=None*, *rebuild_strict=True*, *allow_input_downcast=None*, *profile=None*, *on_unused_input='raise'*)

Return a callable object that will calculate *outputs* from *inputs*.

Parameters

- **params** (*list of either Variable or Param instances, but not shared variables.*) – the returned `Function` instance will have parameters for these variables.
- **outputs** (*list of Variables or Out instances*) – expressions to compute.
- **mode** (None, string or `Mode` instance.) – compilation mode
- **updates** (*iterable over pairs (shared_variable, new_expression). List, tuple or dict.*) – expressions for new `SharedVariable` values
- **givens** (*iterable over pairs (Var1, Var2) of Variables. List, tuple or dict. The Var1 and Var2 in each pair must have the same Type.*) – specific substitutions to make in the computation graph (Var2 replaces Var1).
- **no_default_updates** (*either bool or list of Variables*) – if True, do not perform any automatic update on Variables. If False (default), perform them all. Else, perform automatic updates on all Variables that are neither in `updates` nor in `no_default_updates`.
- **name** – an optional name for this function. The profile mode will print the time spent in this function.
- **rebuild_strict** – True (Default) is the safer and better tested setting, in which case *givens* must substitute new variables with the same Type as the variables they replace. False is a you-better-know-what-you-are-doing setting, that permits *givens* to replace variables with new variables of any Type. The consequence of changing a Type is that all results depending on that variable may have a different Type too (the graph is rebuilt from inputs to outputs). If one of the new types does not make sense for one of the Ops in the graph, an Exception will be raised.
- **allow_input_downcast** (*Boolean or None*) – True means that the values passed as inputs when calling the function can be silently downcasted to fit the dtype of the corresponding Variable, which may lose precision. False means that it will only be cast to a more general, or precise, type. None (default) is almost like False, but allows downcasting of Python float scalars to floatX.
- **profile** (*None, True, or ProfileStats instance*) – accumulate profiling information into a given ProfileStats instance. If argument is *True* then a new ProfileStats instance will be used. This profiling object will be available via self.profile.
- **on_unused_input** – What to do if a variable in the 'inputs' list is not used in the graph. Possible values are 'raise', 'warn', and 'ignore'.

Return type Function instance

Returns a callable object that will compute the outputs (given the inputs) and update the implicit function arguments according to the *updates*.

Inputs can be given as variables or Param instances. `Param` instances also have a variable, but they attach some extra information about how call-time arguments corresponding to that variable should be used. Similarly, `Out` instances can attach information about how output variables should be returned.

The default is typically 'FAST_RUN' but this can be changed in *theano.config*. The mode argument controls the sort of optimizations that will be applied to the graph, and the way the optimized graph will be evaluated.

After each function evaluation, the *updates* mechanism can replace the value of any SharedVariable [implicit] inputs with new values computed from the expressions in the *updates* list. An exception will be raised if you give two update expressions for the same SharedVariable input (that doesn't make sense).

If a SharedVariable is not given an update expression, but has a `default_update` member containing an expression, this expression will be used as the update expression for this variable. Passing `no_default_updates=True` to `function` disables this behavior entirely, passing `no_default_updates=[sharedvar1, sharedvar2]` disables it for the mentioned variables.

Regarding givens: Be careful to make sure that these substitutions are independent, because behaviour when Var1 of one pair appears in the graph leading to Var2 in another expression is undefined (e.g. with `{a: x, b: a + 1}`). Replacements specified with givens are different from optimizations in that Var2 is not expected to be equivalent to Var1.

`theano.compile.function.`**`function_dump`**(*filename*, *inputs*, *outputs=None*, *mode=None*, *updates=None*, *givens=None*, *no_default_updates=False*, *accept_inplace=False*, *name=None*, *rebuild_strict=True*, *allow_input_downcast=None*, *profile=None*, *on_unused_input=None*)

This is helpful to make a reproducable case for problem during Theano compilation.

Ex:

replace *theano.function(...)* by *theano.function_dump('filename.pkl', ...)*.

If you see this, you where probably asked to use this function to help debug a particular case during the compilation of a Theano function. *function_dump* allows to easily reproduce your compilation without asking any code. It pickle all the objects and parameters needed to reproduce a call to *theano.function()*. This include shared variables and there values. If you do not want that, you can set to replace shared variables values by zeros by calling set_value(...) on them before calling *function_dump*.

To load such a dump and do the compilation:

```
>>> import cPickle, theano
>>> d=cPickle.load(open("func_dump.bin", "rb"))
>>> f=theano.function(**d)
```

Note: ***TODO*** Freshen up this old documentation

`io` - defines theano.function [TODO]

Inputs The `inputs` argument to `theano.function` is a list, containing the `Variable` instances for which values will be specified at the time of the function call. But inputs can be more than just Variables. `In` instances let us attach properties to `Variables` to tell function more about how to use them.

class `io.`**`In`**(*object*)

`__init__`(*variable*, *name=None*, *value=None*, *update=None*, *mutable=False*, *strict=False*, *autoname=True*, *implicit=None*)

`variable`: a Variable instance. This will be assigned a value before running the function, not computed from its owner.

`name`: Any type. (If `autoname_input==True`, defaults to `variable.name`). If `name` is a valid Python identifier, this input can be set by `kwarg`, and its value can be accessed by `self.<name>`. The default value is `None`.

`value`: literal or `Container`. The initial/default value for this input. If update is`` None``, this input acts just like an argument with a default value in Python. If update is not `None`, changes to this value will "stick around", whether due to an update or a user's explicit action.

`update`: Variable instance. This expression Variable will replace `value` after each function call. The default value is `None`, indicating that no update is to be done.

`mutable`: Bool (requires value). If `True`, permit the compiled function to modify the Python object being used as the default value. The default value is `False`.

`strict`: Bool (default: `False`). `True` means that the value you pass for this input must have exactly the right type. Otherwise, it may be cast automatically to the proper type.

`autoname`: Bool. If set to `True`, if `name` is `None` and the Variable has a name, it will be taken as the input's name. If autoname is set to `False`, the name is the exact value passed as the name parameter (possibly `None`).

`implicit`: Bool or `None` (default: `None`) `True`: This input is implicit in the sense that the user is not allowed to provide a value for it. Requires `value` to be set.

`False`: The user can provide a value for this input. Be careful when `value` is a container, because providing an input value will overwrite the content of this container.

`None`: Automatically choose between `True` or `False` depending on the situation. It will be set to `False` in all cases except if `value` is a container (so that there is less risk of accidentally overwriting its content without being aware of it).

Value: initial and default values A non-None *value* argument makes an In() instance an optional parameter of the compiled function. For example, in the following code we are defining an arity-2 function `inc`.

```
>>> u, x, s = T.scalars('u', 'x', 's')
>>> inc = function([u, In(x, value=3), In(s, update=(s+x*u), value=10.0)], [])
```

Since we provided a `value` for `s` and `x`, we can call it with just a value for `u` like this:

```
>>> inc(5)          # update s with 10+3*5
[]
>>> print inc[s]
25.0
```

The effect of this call is to increment the storage associated to `s` in `inc` by 15.

If we pass two arguments to `inc`, then we override the value associated to `x`, but only for this one function call.

```
>>> inc(3, 4)       # update s with 25 + 3*4
[]
>>> print inc[s]
37.0
>>> print inc[x]    # the override value of 4 was only temporary
3.0
```

If we pass three arguments to `inc`, then we override the value associated with `x` and `u` and `s`. Since `s`'s value is updated on every call, the old value of `s` will be ignored and then replaced.

```
>>> inc(3, 4, 7)       # update s with 7 + 3*4
[]
>>> print inc[s]
19.0
```

We can also assign to `inc[s]` directly:

```
>>> inc[s] = 10
>>> inc[s]
array(10.0)
```

Input Argument Restrictions The following restrictions apply to the inputs to `theano.function`:

- Every input list element must be a valid `In` instance, or must be upgradable to a valid `In` instance. See the shortcut rules below.
- The same restrictions apply as in Python function definitions: default arguments and keyword arguments must come at the end of the list. Un-named mandatory arguments must come at the beginning of the list.
- Names have to be unique within an input list. If multiple inputs have the same name, then the function will raise an exception. [***Which exception?**]
- Two `In` instances may not name the same Variable. I.e. you cannot give the same parameter multiple times.

If no name is specified explicitly for an In instance, then its name will be taken from the Variable's name. Note that this feature can cause harmless-looking input lists to not satisfy the two conditions above. In such cases, Inputs should be named explicitly to avoid problems such as duplicate names, and named arguments preceding unnamed ones. This automatic naming feature can be disabled by instantiating an In instance explicitly with the `autoname` flag set to False.

Access to function values and containers For each input, `theano.function` will create a `Container` if `value` was not already a `Container` (or if `implicit` was `False`). At the time of a function call, each of these containers must be filled with a value. Each input (but especially ones with a default value or an update expression) may have a value between calls. The function interface defines a way to get at both the current value associated with an input, as well as the container which will contain all future values:

- The `value` property accesses the current values. It is both readable and writable, but assignments (writes) may be implemented by an internal copy and/or casts.
- The `container` property accesses the corresponding container. This property accesses is a read-only dictionary-like interface. It is useful for fetching the container associated with a particular input to share containers between functions, or to have a sort of pointer to an always up-to-date value.

Both `value` and `container` properties provide dictionary-like access based on three types of keys:

- integer keys: you can look up a value/container by its position in the input list;
- name keys: you can look up a value/container by its name;
- Variable keys: you can look up a value/container by the Variable it corresponds to.

In addition to these access mechanisms, there is an even more convenient method to access values by indexing a Function directly by typing `fn[<name>]`, as in the examples above.

To show some examples of these access methods...

```
a, b, c = T.scalars('xys') # set the internal names of graph nodes
# Note that the name of c is 's', not 'c'!
fn = function([a, b, ((c, c+a+b), 10.0)], [])

#the value associated with c is accessible in 3 ways
assert fn['s'] is fn.value[c]
assert fn['s'] is fn.container[c].value

assert fn['s'] == 10.0
fn(1, 2)
assert fn['s'] == 13.0
fn.s = 99.0
fn(1, 0)
assert fn['s'] == 100.0
fn.value[c] = 99.0
fn(1,0)
assert fn['s'] == 100.0
assert fn['s'] == fn.value[c]
assert fn['s'] == fn.container[c].value
```

Input Shortcuts Every element of the inputs list will be upgraded to an In instance if necessary.

- a Variable instance `r` will be upgraded like `In(r)`
- a tuple `(name, r)` will be `In(r, name=name)`
- a tuple `(r, val)` will be `In(r, value=value, autoname=True)`

- a tuple `((r,up), val)` will be `In(r, value=value, update=up, autoname=True)`
- a tuple `(name, r, val)` will be `In(r, name=name, value=value)`
- a tuple `(name, (r,up), val)` will be `In(r, name=name, value=val, update=up, autoname=True)`

Example:

```
import theano
from theano import tensor as T
from theano.compile.io import In
x = T.scalar()
y = T.scalar('y')
z = T.scalar('z')
w = T.scalar('w')

fn = theano.function(inputs = [x, y, In(z, value=42), ((w, w+x), 0)],
                     outputs = x + y + z)
# the first two arguments are required and the last two are
# optional and initialized to 42 and 0, respectively.
# The last argument, w, is updated with w + x each time the
# function is called.

fn(1)               # illegal because there are two required arguments
fn(1, 2)            # legal, z is 42, w goes 0 -> 1 (because w <- w + x), returns array(45
fn(1, y = 2)        # legal, z is 42, w goes 1 -> 2, returns array(45.0)
fn(x = 1, y = 2)    # illegal because x was not named
fn(1, 2, 3)         # legal, z is 3, w goes 2 -> 3, returns array(6.0)
fn(1, z = 3, y = 2) # legal, z is 3, w goes 3 -> 4, returns array(6.0)
fn(1, 2, w = 400)   # legal, z is 42 again, w goes 400 -> 401, returns array(45.0)
fn(1, 2)            # legal, z is 42, w goes 401 -> 402, returns array(45.0)
```

In the example above, `z` has value 42 when no value is explicitly given. This default value is potentially used at every function invocation, because `z` has no `update` or storage associated with it.

Outputs The `outputs` argument to function can be one of

- `None`, or
- a Variable or `Out` instance, or
- a list of Variables or `Out` instances.

An `Out` instance is a structure that lets us attach options to individual output `Variable` instances, similarly to how `In` lets us attach options to individual input `Variable` instances.

Out(variable, borrow=False) returns an `Out` instance:

- `borrow`

  If `True`, a reference to function's internal storage is OK. A value returned for this output might be clobbered by running the function again, but the function might be faster.

  Default: `False`

If a single `Variable` or `Out` instance is given as argument, then the compiled function will return a single value.

If a list of `Variable` or `Out` instances is given as argument, then the compiled function will return a list of their values.

```
x, y, s = T.matrices('xys')

# print a list of 2 ndarrays
fn1 = theano.function([x], [x+x, Out((x+x).T, borrow=True)])
print fn1(numpy.asarray([[1,0],[0,1]]))


# print a list of 1 ndarray
fn2 = theano.function([x], [x+x])
print fn2(numpy.asarray([[1,0],[0,1]]))

# print an ndarray
fn3 = theano.function([x], outputs=x+x)
print fn3(numpy.asarray([[1,0],[0,1]]))
```

`ops` – Some Common Ops and extra Ops stuff

This file contains auxiliary Ops, used during the compilation phase and Ops building class (`FromFunctionOp`) and decorator (`as_op()`) that help make new Ops more rapidly.

class `theano.compile.ops.`**`FromFunctionOp`**(*fn*, *itypes*, *otypes*, *infer_shape*)
Build a basic Theano Op around a function.

Since the resulting Op is very basic and is missing most of the optional functionalities, some optimizations may not apply. If you want to help, you can supply an infer_shape function that computes the shapes of the output given the shapes of the inputs.

Also the gradient is undefined in the resulting op and Theano will raise an error if you attempt to get the gradient of a graph containing this op.

class `theano.compile.ops.`**`OutputGuard`**(*use_c_code='/usr/bin/g++'*)
This op is used only internally by Theano.

Only the AddDestroyHandler optimizer tries to insert them in the graph.

This Op is declared as destructive while it is not destroying anything. It returns a view. This is used to prevent destruction of the output variables of a Theano function.

There is a mechanism in Theano that should prevent this, but the use of OutputGuard adds a safeguard: it may be possible for some optimization run before the add_destroy_handler phase to bypass this mechanism, by making in-place optimizations.

TODO: find a current full explanation.

class `theano.compile.ops.`**`Rebroadcast`**(**axis*)
Change the input's broadcastable fields in some predetermined way.

unbroadcast <theano.tensor.unbroadcast> addbroadcast <theano.tensor.addbroadcast> patternbroadcast <theano.tensor.patternbroadcast>

Works inplace and works for CudaNdarrayType.

Rebroadcast((0, True), (1, False))(x) would make *x* broadcastable in axis 0 and not broadcastable in axis 1.

class theano.compile.ops.**Shape**(*use_c_code='/usr/bin/g++'*)
L{Op} to return the shape of a matrix.

Non-differentiable.

class theano.compile.ops.**Shape_i**(*i*)
L{Op} to return the shape of a matrix.

Non-differentiable.

class theano.compile.ops.**SpecifyShape**(*use_c_code='/usr/bin/g++'*)
L{Op} that puts into the graph the user-provided shape.

In the case where this op stays in the final graph, we assert the shape. For this the output of this op must be used in the graph. This is not the case most of the time if we only take the shape of the output. Maybe there are other optimizations that will mess with this.

Maybe in the future we will never do the assert!

We currently don't support specifying partial shape information.

TODO : test this op with sparse and cuda ndarray. Do C code for them too.

class theano.compile.ops.**ViewOp**(*use_c_code='/usr/bin/g++'*)
Returns an inplace view of the input. Used internally by Theano.

theano.compile.ops.**as_op**(*itypes*, *otypes*, *infer_shape=None*)
Decorator that converts a function into a basic Theano op that will call the supplied function as its implementation.

It takes an optional infer_shape parameter that should be a callable with this signature:

def infer_shape(node, input_shapes): ... return output_shapes

Here *input_shapes* and *output_shapes* are lists of tuples that represent the shape of the corresponding inputs/outputs.

This should not be used when performance is a concern since the very basic nature of the resulting Op may interfere with certain graph optimizations.

@as_op(itypes=[theano.tensor.fmatrix, theano.tensor.fmatrix], otypes=[theano.tensor.fmatrix])

def numpy_dot(a, b): return numpy.dot(a, b)

theano.compile.ops.**register_deep_copy_op_c_code**(*typ*, *code*, *version=()*)
Tell DeepCopyOp how to generate C code for a Theano Type.

typ [Theano type] It must be the Theano class itself and not an instance of the class.

code: C code Deep copies the Theano type 'typ'. Use %(iname)s and %(oname)s for the input and output C variable names respectively.

version A number indicating the version of the code, for cache.

theano.compile.ops.**register_rebroadcast_c_code**(*typ*, *code*, *version=()*)
Tell Rebroadcast how to generate C code for a Theano Type.

typ [Theano type] It must be the Theano class itself and not an instance of the class.

code [C code] That checks if the dimension %(axis)s is of shape 1 for the Theano type 'typ'. Use %(iname)s and %(oname)s for the input and output C variable names respectively, and %(axis)s for the axis that we need to check. This code is put in a loop for all axes.

version A number indicating the version of the code, for cache.

theano.compile.ops.**register_shape_c_code**(*type*, *code*, *version=()*)
Tell Shape Op how to generate C code for a Theano Type.

typ [Theano type] It must be the Theano class itself and not an instance of the class.

code [C code] Returns a vector representing the shape for the Theano type 'typ'. Use %(iname)s and %(oname)s for the input and output C variable names respectively.

version A number indicating the version of the code, for cache.

theano.compile.ops.**register_shape_i_c_code**(*typ*, *code*, *check_input*, *version=()*)
Tell Shape_i how to generate C code for a Theano Type.

typ [Theano type] It must be the Theano class itself and not an instance of the class.

code [C code] Gets the shape of dimensions %(i)s for the Theano type 'typ'. Use %(iname)s and %(oname)s for the input and output C variable names respectively.

version A number indicating the version of the code, for cache.

theano.compile.ops.**register_specify_shape_c_code**(*typ*, *code*, *version=()*, *c_support_code_apply=None*)
Tell SpecifyShape how to generate C code for a Theano Type.

typ [Theano type] It must be the Theano class itself and not an instance of the class.

code [C code] Checks the shape and returns a view for the Theano type 'typ'. Use %(iname)s and %(oname)s for the input and output C variable names respectively. %(shape)s is the vector of shape of %(iname)s. Check that its length is good.

version A number indicating the version of the code, for cache.

c_support_code_apply Extra code.

theano.compile.ops.**register_view_op_c_code**(*type*, *code*, *version=()*)
Tell ViewOp how to generate C code for a Theano Type.

type [Theano type] It must be the Theano class itself and not an instance of the class.

code [C code] Returns a view for the Theano type 'type'. Use %(iname)s and %(oname)s for the input and output C variable names respectively.

version A number indicating the version of the code, for cache.

theano.compile.ops.**shape_i**(*var*, *i*, *fgraph=None*)
Equivalent of var.shape[i], but apply if possible the shape feature optimization.

This is useful in optimization that need to get the shape. This remove the need of the following shape_feature optimization that convert it. So this speed up optimization and remove Equilibrium max iteration problems.

var The variable we want to take the shape of.

i The shape dimensions we want

fgraph [optional] If var.fgraph do not exist, the fgraph that have the shape_feature to introduce var in to get the optimized shape.

`mode` – controlling compilation

Guide The `mode` parameter to `theano.function()` controls how the inputs-to-outputs graph is transformed into a callable object.

Theano defines the following modes by name:

- `'FAST_COMPILE'`: Apply just a few graph optimizations and only use Python implementations.
- `'FAST_RUN'`: Apply all optimizations, and use C implementations where possible.
- `'DebugMode'`: A mode for debuging. See *DebugMode* for details.
- `'ProfileMode'`: Deprecated, use the Theano flag `config.profile`.
- `'DEBUG_MODE'`: Deprecated. Use the string DebugMode.
- `'PROFILE_MODE'`: Deprecated. Use the string ProfileMode.

The default mode is typically `FAST_RUN`, but it can be controlled via the configuration variable `config.mode`, which can be overridden by passing the keyword argument to `theano.function()`.

Todo

For a finer level of control over which optimizations are applied, and whether C or Python implementations are used, read.... what exactly?

Reference

`mode.`**`FAST_COMPILE`**
`mode.`**`FAST_RUN`**

class `mode.`**`Mode`**(*object*)

Compilation is controlled by two attributes: the *optimizer* controls how an expression graph will be transformed; the *linker* controls how the optimized expression graph will be evaluated.

`optimizer`
An `optimizer` instance.

`linker`
A `linker` instance.

including(**tags*)
Return a new Mode instance like this one, but with an optimizer modified by including the given tags.

excluding(**tags*)
Return a new Mode instance like this one, but with an optimizer modified by excluding the given tags.

requiring(**tags*)
Return a new Mode instance like this one, but with an optimizer modified by requiring the given tags.

debugmode

Guide The DebugMode evaluation mode includes a number of self-checks and assertions that can help to diagnose several kinds of programmer errors that can lead to incorrect output.

It is much slower to evaluate a function or method with DebugMode than it would be in `'FAST_RUN'` or even `'FAST_COMPILE'`. We recommended you use DebugMode during development, but not when you launch 1000 processes on a cluster.

DebugMode can be used as follows:

```
x = tensor.dscalar('x')

f = theano.function([x], 10*x, mode='DebugMode')

f(5)
f(0)
f(7)
```

It can also be used by setting the configuration variable `config.mode`. It can also be used by passing a DebugMode instance as the mode, as in

```
>>> f = theano.function([x], 10*x, mode=DebugMode(check_c_code=False))
```

If any problem is detected, DebugMode will raise an exception according to what went wrong, either at call time (`f(5)`) or compile time (`f = theano.function(x, 10*x, mode='DebugMode')`). These exceptions should *not* be ignored; talk to your local Theano guru or email the users list if you cannot make the exception go away.

Some kinds of errors can only be detected for certain input value combinations. In the example above, there is no way to guarantee that a future call to say, `f(-1)` won't cause a problem. DebugMode is not a silver bullet.

If you instantiate DebugMode using the constructor `compile.DebugMode` rather than the keyword `DebugMode` you can configure its behaviour via constructor arguments.

Reference

class `debugmode.`**DebugMode**(*Mode*)
Evaluation Mode that detects internal theano errors.

This mode catches several kinds of internal error:

- inconsistent outputs when calling the same Op twice with the same inputs, for instance if c_code and perform implementations, are inconsistent, or in case of incorrect handling of output memory (see *BadThunkOutput*)
- a variable replacing another when their runtime values don't match. This is a symptom of an incorrect optimization step, or faulty Op implementation (raises *BadOptimization*)
- stochastic optimization ordering (raises *StochasticOrder*)
- incomplete *destroy_map* specification (raises *BadDestroyMap*)
- an op that returns an illegal value not matching the output Variable Type (raises InvalidValueError)

Each of these exceptions inherits from the more generic *DebugModeError*.

If there are no internal errors, this mode behaves like FAST_RUN or FAST_COMPILE, but takes a little longer and uses more memory.

If there are internal errors, this mode will raise an *DebugModeError* exception.

stability_patience = config.DebugMode.patience
When checking for the stability of optimization, recompile the graph this many times. Default 10.

check_c_code = config.DebugMode.check_c
Should we evaluate (and check) the *c_code* implementations?

`True` -> yes, `False` -> no.

Default yes.

check_py_code = config.DebugMode.check_py

Should we evaluate (and check) the *perform* implementations?

`True` -> yes, `False` -> no.

Default yes.

check_isfinite = config.DebugMode.check_finite
Should we check for (and complain about) `NaN`/`Inf` ndarray elements?

`True` -> yes, `False` -> no.

Default yes.

require_matching_strides = config.DebugMode.check_strides
Check for (and complain about) Ops whose python and C outputs are ndarrays with different strides. (This can catch bugs, but is generally overly strict.)

0 -> no check, 1 -> warn, 2 -> err.

Default warn.

__init__(*self*, *optimizer='fast_run'*, *stability_patience=None*, *check_c_code=None*, *check_py_code=None*, *check_isfinite=None*, *require_matching_strides=None*, *linker=None*)

Initialize member variables.

If any of these arguments (except optimizer) is not None, it overrides the class default. The linker arguments is not used. It is set their to allow Mode.requiring() and some other fct to work with DebugMode too.

The keyword version of DebugMode (which you get by using `mode='DebugMode`) is quite strict, and can raise several different Exception types. There following are DebugMode exceptions you might encounter:

class debugmode.**DebugModeError**(*Exception*)

This is a generic error. All the other exceptions inherit from this one. This error is typically not raised directly. However, you can use `except DebugModeError: ...` to catch any of the more specific types of Exception.

class debugmode.**BadThunkOutput**(*DebugModeError*)

This exception means that different calls to the same Op with the same inputs did not compute the same thing like they were supposed to. For instance, it can happen if the python (`perform`) and c (`c_code`) implementations of the Op are inconsistent (the problem might be a bug in either `perform` or `c_code` (or both)). It can also happen if `perform` or `c_code` does not handle correctly output memory that has been preallocated (for instance, if it did not clear the memory before accumulating into it, or if it assumed the memory layout was C-contiguous even if it is not).

class debugmode.**BadOptimization**(*DebugModeError*)

This exception indicates that an Optimization replaced one variable (say V1) with another one (say V2) but at runtime, the values for V1 and V2 were different. This is something that optimizations are not supposed to do.

It can be tricky to identify the one-true-cause of an optimization error, but this exception provides a lot of guidance. Most of the time, the exception object will indicate which optimization was at fault. The exception object also contains information such as a snapshot of the before/after graph where the optimization introduced the error.

class debugmode.**BadDestroyMap**(*DebugModeError*)

This happens when an Op's `perform()` or `c_code()` modifies an input that it wasn't supposed to. If either the `perform` or `c_code` implementation of an Op might modify any input, it has to advertise that fact via the `destroy_map` attribute.

For detailed documentation on the `destroy_map` attribute, see *Inplace operations*.

class debugmode.**BadViewMap**(*DebugModeError*)

This happens when an Op's perform() or c_code() creates an alias or alias-like dependency between an input and an output... and it didn't warn the optimization system via the `view_map` attribute.

For detailed documentation on the `view_map` attribute, see *Views*.

class debugmode.**StochasticOrder**(*DebugModeError*)

This happens when an optimization does not perform the same graph operations in the same order when run several times in a row. This can happen if any steps are ordered by `id(object)` somehow, such as via the default object hash function. A Stochastic optimization invalidates the pattern of work whereby we debug in DebugMode and then run the full-size jobs in FAST_RUN.

class debugmode.**InvalidValueError**(*DebugModeError*)

This happens when some Op's perform or c_code implementation computes an output that is invalid with respect to the type of the corresponding output variable. Like if it returned a complex-valued ndarray for a dscalar Type.

This can also be triggered when floating-point values such as NaN and Inf are introduced into the computations. It indicates which Op created the first NaN. These floating-point values can be allowed by passing the check_isfinite=False argument to DebugMode.

profilemode – profiling Theano functions

Guide

Note: ProfileMode is deprecated. Use config.profile instead.

To profile a Theano graph, a special mode called ProfileMode, must be passed as an argument when compiling your graph. Using ProfileMode is a three-step process.

Creating a ProfileMode Instance First create a ProfileMode instance.

```
>>> from theano import ProfileMode
>>> profmode = theano.ProfileMode(optimizer='fast_run', linker=theano.gof.OpWiseCLinker())
```

The ProfileMode constructor takes as input an optimizer and a linker. Which optimizer and linker to use will depend on the application. For example, a user wanting to profile the Python implementation only, should use the gof.PerformLinker (or "py" for short). On the other hand, a user wanting to profile his graph using C implementations wherever possible should use the gof.OpWiseCLinker (or "c|py").

In the same manner, modifying which optimizer is passed to ProfileMode will decide which optimizations are applied to the graph, prior to profiling. Changing the optimizer should be especially useful when developing new graph optimizations, in order to evaluate their impact on performance. Also keep in mind that optimizations might change the computation graph a lot, meaning that you might not recognize some of the operations that are profiled (you did not use them explicitly but an optimizer decided to use it to improve performance or numerical stability). If you cannot easily relate the output of ProfileMode with the computations you defined, you might want to try setting optimizer to None (but keep in mind the computations will be slower than if they were optimized).

Note that most users will want to use ProfileMode to optimize their graph and find where most of the computation time is being spent. In this context, 'fast_run' optimizer and gof.OpWiseCLinker are the most appropriate choices.

Compiling your Graph with ProfileMode Once the ProfileMode instance is created, simply compile your graph as you would normally, by specifying the mode parameter.

```
>>> # with functions
>>> f = theano.function([input1,input2],[output1], mode=profmode)
```

Retrieving Timing Information Once your graph is compiled, simply run the program or operation you wish to profile, then call `profmode.print_summary()`. This will provide you with the desired timing information, indicating where your graph is spending most of its time.

This is best shown through an example. Lets use the example of logistic regression. (Code for this example is in the file `benchmark/regression/regression.py`.)

Compiling the module with ProfileMode and calling `profmode.print_summary()` generates the following output:

```
"""
ProfileMode.print_summary()
---------------------------

local_time 0.0749197006226 (Time spent running thunks)
Apply-wise summary: <fraction of local_time spent at this position> (<Apply position>, <App
        0.069   15      _dot22
        0.064   1       _dot22
        0.053   0       InplaceDimShuffle{x,0}
        0.049   2       InplaceDimShuffle{1,0}
        0.049   10      mul
        0.049   6       Elemwise{ScalarSigmoid{output_types_preference=<theano.scalar.basi
        0.049   3       InplaceDimShuffle{x}
        0.049   4       InplaceDimShuffle{x,x}
        0.048   14      Sum{0}
        0.047   7       sub
        0.046   17      mul
        0.045   9       sqr
        0.045   8       Elemwise{sub}
        0.045   16      Sum
        0.044   18      mul
   ... (remaining 6 Apply instances account for 0.25 of the runtime)
Op-wise summary: <fraction of local_time spent on this kind of Op> <Op name>
        0.139   * mul
        0.134   * _dot22
        0.092   * sub
        0.085   * Elemwise{Sub{output_types_preference=<theano.scalar.basic.transfer_type
        0.053   * InplaceDimShuffle{x,0}
        0.049   * InplaceDimShuffle{1,0}
        0.049   * Elemwise{ScalarSigmoid{output_types_preference=<theano.scalar.basic.tran
        0.049   * InplaceDimShuffle{x}
        0.049   * InplaceDimShuffle{x,x}
        0.048   * Sum{0}
        0.045   * sqr
        0.045   * Sum
        0.043   * Sum{1}
        0.042   * Elemwise{Mul{output_types_preference=<theano.scalar.basic.transfer_type
        0.041   * Elemwise{Add{output_types_preference=<theano.scalar.basic.transfer_type
        0.039   * Elemwise{Second{output_types_preference=<theano.scalar.basic.transfer_ty
   ... (remaining 0 Ops account for 0.00 of the runtime)
(*) Op is running a c implementation

"""
```

Note: *TODO*

The following text was recovered from a recent version of the source file... hopefully things haven't gotten too out-of-sync!

The first show an Apply-wise summary, the second show an Op-wise summary, the third show an type-Op-wise summary.

The Apply-wise summary print the timing information for the worst offending Apply nodes. This corresponds to individual Op applications within your graph which take the longest to execute (so if you use dot twice, you will see two entries there).

The Op-wise summary print the execution time of all Apply nodes executing the same Op are grouped together and the total execution time per Op is shown (so if you use dot twice, you will see only one entry there corresponding to the sum of the time spent in each of them). If two Op have different hash value, they will be separate.

The type-Op-wise summary group the result by type of op. So event if two Op have different hash value, they will be merged.

Their is an hack with the Op-wise summary. Go see it if you want to know more.

The summary has two components to it. In the first section called the Apply-wise summary, timing information is provided for the worst offending Apply nodes. This corresponds to individual Op applications within your graph which take the longest to execute (so if you use `dot` twice, you will see two entries there). In the second portion, the Op-wise summary, the execution time of all Apply nodes executing the same Op are grouped together and the total execution time per Op is shown (so if you use `dot` twice, you will see only one entry there corresponding to the sum of the time spent in each of them).

Note that the ProfileMode also shows which Ops were running a c implementation.

Developers wishing to optimize the performance of their graph should focus on the worst offending Ops and Apply nodes – either by optimizing an implementation, providing a missing C implementation, or by writing a graph optimization that eliminates the offending Op altogether. You should strongly consider emailing one of our lists about your issue before spending too much time on this.

Reference

class `profilemode.`**`ProfileMode`**(*Mode*)

`print_summary`(*n_apply_to_print=None*, *n_ops_to_print=None*)

Print three summaries to stdout that show where cpu time is spent during theano function executions (for all functions using this object instance).

Parameters

- **n_apply_to_print** – the number of apply nodes to print. The default 15, but can be configured via `ProfileMode.n_ops_to_print` in `THEANO_FLAGS`.
- **n_ops_to_print** – the number of ops to print. Default 20, or but can be configured via `ProfileMode.n_apply_to_print` in `THEANO_FLAGS`.

Returns None

print_diff_summary(self, other, n_apply_to_print=None, n_ops_to_print=None):
""" As print_summary, but print the difference on two different profile mode.
TODO: Also we don't print the Apply-wise summary as it don't work for now.
TODO: make comparaison with gpu code.

Parameters

- **other** – the other instance of ProfileMode that we want to be compared to.
- **n_apply_to_print** – the number of apply nodes to print. The default 15, but can be configured via `ProfileMode.n_ops_to_print` in `THEANO_FLAGS`.
- **n_ops_to_print** – the number of ops to print. Default 20, or but can be configured via `ProfileMode.n_apply_to_print` in `THEANO_FLAGS`.

Returns None

nanguardmode

Guide The NanGuardMode aims to prevent the model from outputing NaNs or Infs. It has a number of self-checks, which can help to find out which apply node is generating those incorrect outputs. It provides automatic detection of 3 types of abnormal values: NaNs, Infs, and abnormally big values.

NanGuardMode can be used as follows:

```
x = T.matrix()
w = theano.shared(numpy.random.randn(5, 7).astype(theano.config.floatX))
y = T.dot(x, w)
fun = theano.function(
    [x], y,
    mode=NanGuardMode(nan_is_error=True, inf_is_error=True, big_is_error=True)
)
```

While using the theano function `fun`, it will monitor the values of each input and output variable of each node. When abnormal values are detected, it raises an error to indicate which node yields the NaNs. For example, if we pass the following values to `fun`:

```
infa = numpy.tile(
    (numpy.asarray(100.) ** 1000000).astype(theano.config.floatX), (3, 5))
fun(infa)
```

It will raise an AssertionError indicating that Inf value is detected while executing the function.

You can also set the three parameters in `NanGuardMode()` to indicate which kind of abnormal values to monitor. `nan_is_error` and `inf_is_error` has no default values, so they need to be set explicitly, but `big_is_error` is set to be `True` by default.

Reference

class `theano.compile.nanguardmode.`**NanGuardMode**(*nan_is_error*, *inf_is_error*, *big_is_error=True*)

A Theano compilation Mode that makes the compiled function automatically detect NaNs and Infs and detect an error if they occur.

nan_is_error [bool] If True, raise an error anytime a NaN is encountered.

inf_is_error [bool] If True, raise an error anytime an Inf is encountered. Note that some pylearn2 modules currently use np.inf as a default value (e.g. mlp.max_pool) and these will cause an error if inf_is_error is True.

big_is_error [bool] If True, raise an error when a value greater than 1e10 is encountered.

`sparse` – Symbolic Sparse Matrices

In the tutorial section, you can find a *sparse tutorial*.

The sparse submodule is not loaded when we import Theano. You must import `theano.sparse` to enable it.

The sparse module provides the same functionality as the tensor module. The difference lies under the covers because sparse matrices do not store data in a contiguous array. Note that there are no GPU implementations for sparse matrices in Theano. The sparse module has been used in:

- NLP: Dense linear transformations of sparse vectors.
- Audio: Filterbank in the Fourier domain.

Compressed Sparse Format

This section tries to explain how information is stored for the two sparse formats of SciPy supported by Theano. There are more formats that can be used with SciPy and some documentation about them may be found here.

Theano supports two *compressed sparse formats* `csc` and `csr`, respectively based on columns and rows. They have both the same attributes: `data`, `indices`, `indptr` and `shape`.

- The `data` attribute is a one-dimentionnal `ndarray` which contains all the non-zero elements of the sparse matrix.
- The `indices` and `indptr` attributes are used to store the position of the data in the sparse matrix.
- The `shape` attribute is exactly the same as the `shape` attribute of a dense (i.e. generic) matrix. It can be explicitly specified at the creation of a sparse matrix if it cannot be infered from the first three attributes.

CSC Matrix In the *Compressed Sparse Column* format, `indices` stands for indexes inside the column vectors of the matrix and `indptr` tells where the column starts in the `data` and in the `indices` attributes. `indptr` can be thought of as giving the slice which must be applied to the other attribute in order to get each column of the matrix. In other words, `slice(indptr[i], indptr[i+1])` corresponds to the slice needed to find the i-th column of the matrix in the `data` and `indices` fields.

The following example builds a matrix and returns its columns. It prints the i-th column, i.e. a list of indices in the column and their corresponding value in the second list.

```
>>> data = np.asarray([7, 8, 9])
>>> indices = np.asarray([0, 1, 2])
>>> indptr = np.asarray([0, 2, 3, 3])
>>> m = sp.csc_matrix((data, indices, indptr), shape=(3, 3))
>>> print m.toarray()
[[7 0 0]
 [8 0 0]
 [0 9 0]]
>>> i = 0
>>> print m.indices[m.indptr[i]:m.indptr[i+1]], m.data[m.indptr[i]:m.indptr[i+1]]
[0, 1] [7, 8]
>>> i = 1
>>> print m.indices[m.indptr[i]:m.indptr[i+1]], m.data[m.indptr[i]:m.indptr[i+1]]
[2] [9]
>>> i = 2
>>> print m.indices[m.indptr[i]:m.indptr[i+1]], m.data[m.indptr[i]:m.indptr[i+1]]
[] []
```

CSR Matrix In the *Compressed Sparse Row* format, `indices` stands for indexes inside the row vectors of the matrix and `indptr` tells where the row starts in the `data` and in the `indices` attributes. `indptr` can be thought of as giving the slice which must be applied to the other attribute in order to get each row of the matrix. In other words, `slice(indptr[i], indptr[i+1])` corresponds to the slice needed to find the i-th row of the matrix in the `data` and `indices` fields.

The following example builds a matrix and returns its rows. It prints the i-th row, i.e. a list of indices in the row and their corresponding value in the second list.

```
>>> data = np.asarray([7, 8, 9])
>>> indices = np.asarray([0, 1, 2])
>>> indptr = np.asarray([0, 2, 3, 3])
>>> m = sp.csr_matrix((data, indices, indptr), shape=(3, 3))
>>> print m.toarray()
[[7 8 0]
 [0 0 9]
 [0 0 0]]
>>> i = 0
>>> print m.indices[m.indptr[i]:m.indptr[i+1]], m.data[m.indptr[i]:m.indptr[i+1]]
[0, 1] [7, 8]
>>> i = 1
>>> print m.indices[m.indptr[i]:m.indptr[i+1]], m.data[m.indptr[i]:m.indptr[i+1]]
[2] [9]
>>> i = 2
>>> print m.indices[m.indptr[i]:m.indptr[i+1]], m.data[m.indptr[i]:m.indptr[i+1]]
[] []
```

List of Implemented Operations

- **Moving from and to sparse**
  - `dense_from_sparse`. Both grads are implemented. Structured by default.

  - csr_from_dense, csc_from_dense. The grad implemented is structured.
  - Theano SparseVariable objects have a method toarray() that is the same as dense_from_sparse.

- **Construction of Sparses and their Properties**
  - CSM and CSC, CSR to construct a matrix. The grad implemented is regular.
  - csm_properties. to get the properties of a sparse matrix. The grad implemented is regular.
  - csm_indices(x), csm_indptr(x), csm_data(x) and csm_shape(x) or x.shape.
  - sp_ones_like. The grad implemented is regular.
  - sp_zeros_like. The grad implemented is regular.
  - square_diagonal. The grad implemented is regular.
  - construct_sparse_from_list. The grad implemented is regular.
- **Cast**
  - cast with bcast, wcast, icast, lcast, fcast, dcast, ccast, and zcast. The grad implemented is regular.
- **Transpose**
  - transpose. The grad implemented is regular.
- **Basic Arithmetic**
  - neg. The grad implemented is regular.
  - eq.
  - neq.
  - gt.
  - ge.
  - lt.
  - le.
  - add. The grad implemented is regular.
  - sub. The grad implemented is regular.
  - mul. The grad implemented is regular.
  - col_scale to multiply by a vector along the columns. The grad implemented is structured.
  - row_slace to multiply by a vector along the rows. The grad implemented is structured.
- **Monoid (Element-wise operation with only one sparse input).** *They all have a structured grad.*
  - structured_sigmoid

  - structured_exp
  - structured_log
  - structured_pow
  - structured_minimum
  - structured_maximum
  - structured_add
  - sin
  - arcsin
  - tan
  - arctan
  - sinh
  - arcsinh
  - tanh
  - arctanh
  - rad2deg
  - deg2rad
  - rint
  - ceil
  - floor
  - trunc
  - sgn
  - log1p
  - expm1
  - sqr
  - sqrt

- **Dot Product**
  - dot.
    - One of the inputs must be sparse, the other sparse or dense.
    - The grad implemented is regular.
    - No C code for perform and no C code for grad.
    - Returns a dense for perform and a dense for grad.
  - structured_dot.

        * The first input is sparse, the second can be sparse or dense.
        * The grad implemented is structured.
        * C code for perform and grad.
        * It returns a sparse output if both inputs are sparse and dense one if one of the inputs is dense.
        * Returns a sparse grad for sparse inputs and dense grad for dense inputs.
    - `true_dot`.
        * The first input is sparse, the second can be sparse or dense.
        * The grad implemented is regular.
        * No C code for perform and no C code for grad.
        * Returns a Sparse.
        * The gradient returns a Sparse for sparse inputs and by default a dense for dense inputs. The parameter `grad_preserves_dense` can be set to False to return a sparse grad for dense inputs.
    - `sampling_dot`.
        * Both inputs must be dense.
        * The grad implemented is structured for *p*.
        * Sample of the dot and sample of the gradient.
        * C code for perform but not for grad.
        * Returns sparse for perform and grad.
    - `usmm`.
        * **You *shouldn't* insert this op yourself!**
            · There is an optimization that transform a `dot` to `Usmm` when possible.
        * This op is the equivalent of gemm for sparse dot.
        * There is no grad implemented for this op.
        * One of the inputs must be sparse, the other sparse or dense.
        * Returns a dense from perform.

- **Slice Operations**
    - sparse_variable[N, N], returns a tensor scalar. There is no grad implemented for this operation.
    - sparse_variable[M:N, O:P], returns a sparse matrix There is no grad implemented for this operation.

  - Sparse variables don't support [M, N:O] and [M:N, O] as we don't support sparse vectors and returning a sparse matrix would break the numpy interface. Use [M:M+1, N:O] and [M:N, O:O+1] instead.
  - `diag`. The grad implemented is regular.

- **Concatenation**
  - `hstack`. The grad implemented is regular.
  - `vstack`. The grad implemented is regular.

- **Probability** *There is no grad implemented for these operations.*
  - `Poisson` and `poisson`
  - `Binomial` and `csc_fbinomial`, `csc_dbinomial` `csr_fbinomial`, `csr_dbinomial`
  - `Multinomial` and `multinomial`

- **Internal Representation** *They all have a regular grad implemented.*
  - `ensure_sorted_indices`.
  - `remove0`.
  - `clean` to resort indices and remove zeros

- **To help testing**
  - `theano.sparse.tests.test_basic.sparse_random_inputs()`

`sparse` – Sparse Op

Classes for handling sparse matrices.

To read about different sparse formats, see http://www-users.cs.umn.edu/~saad/software/SPARSKIT/paper.ps

`theano.sparse.basic.`**`add`**(*x*, *y*)

Add two matrices, at least one of which is sparse.

This method will provide the right op according to the inputs.

x A matrix variable.

y A matrix variable.

matrix $x + y$

At least one of *x* and *y* must be a sparse matrix.

The grad will be structured only when one of the variable will be a dense matrix.

theano.sparse.basic.**as_sparse**(*x*, *name=None*)

Wrapper around SparseVariable constructor to construct a Variable with a sparse matrix with the same dtype and format.

x A sparse matrix.

object SparseVariable version of *x*.

theano.sparse.basic.**as_sparse_or_tensor_variable**(*x*, *name=None*)

Same as *as_sparse_variable* but if we can't make a sparse variable, we try to make a tensor variable.

x A sparse matrix.

SparseVariable or TensorVariable version of *x*

theano.sparse.basic.**as_sparse_variable**(*x*, *name=None*)

Wrapper around SparseVariable constructor to construct a Variable with a sparse matrix with the same dtype and format.

x A sparse matrix.

object SparseVariable version of *x*.

theano.sparse.basic.**cast**(*variable*, *dtype*)

Cast sparse variable to the desired dtype.

variable Sparse matrix.

dtype The dtype wanted.

Same as *x* but having *dtype* as dtype.

The grad implemented is regular, i.e. not structured.

theano.sparse.basic.**clean**(*x*)

Remove explicit zeros from a sparse matrix, and re-sort indices.

CSR column indices are not necessarily sorted. Likewise for CSC row indices. Use *clean* when sorted indices are required (e.g. when passing data to other libraries) and to ensure there are no zeros in the data.

x A sparse matrix.

matrix The same as *x* with indices sorted and zeros removed.

The grad implemented is regular, i.e. not structured.

theano.sparse.basic.**col_scale**(*x*, *s*)

Scale each columns of a sparse matrix by the corresponding element of a dense vector.

x A sparse matrix.

s A dense vector with length equal to the number of columns of *x*.

A sparse matrix in the same format as *x* which each column had been multiply by the corresponding element of *s*.

The grad implemented is structured.

theano.sparse.basic.**csm_data**(*csm*)
Return the data field of the sparse variable.

theano.sparse.basic.**csm_indices**(*csm*)
Return the indices field of the sparse variable.

theano.sparse.basic.**csm_indptr**(*csm*)
Return the indptr field of the sparse variable.

theano.sparse.basic.**csm_shape**(*csm*)
Return the shape field of the sparse variable.

theano.sparse.basic.**dot**(*x*, *y*)
Operation for efficiently calculating the dot product when one or all operands is sparse. Supported format are CSC and CSR. The output of the operation is dense.

x Sparse or dense matrix variable.

y Sparse or dense matrix variable.

The dot product *x*.'y' in a dense format.

The grad implemented is regular, i.e. not structured.

At least one of *x* or *y* must be a sparse matrix.

When the operation has the form dot(csr_matrix, dense) the gradient of this operation can be performed inplace by UsmmCscDense. This leads to significant speed-ups.

theano.sparse.basic.**hstack**(*blocks*, *format=None*, *dtype=None*)
Stack sparse matrices horizontally (column wise).

This wrap the method hstack from scipy.

blocks List of sparse array of compatible shape.

format String representing the output format. Default is csc.

dtype Output dtype.

array The concatenation of the sparse array column wise.

The number of line of the sparse matrix must agree.

The grad implemented is regular, i.e. not structured.

theano.sparse.basic.**mul**(*x*, *y*)
Multiply elementwise two matrices, at least one of which is sparse.

This method will provide the right op according to the inputs.

x A matrix variable.

y A matrix variable.

matrix $x + y$

At least one of x and y must be a sparse matrix. The grad is regular, i.e. not structured.

theano.sparse.basic.**row_scale**(x, s)

Scale each row of a sparse matrix by the corresponding element of a dense vector.

x A sparse matrix.

s A dense vector with length equal to the number of rows of x.

matrix A sparse matrix in the same format as x whose each row has been multiplied by the corresponding element of s.

The grad implemented is structured.

theano.sparse.basic.**sp_ones_like**(x)

Construct a sparse matrix of ones with the same sparsity pattern.

x Sparse matrix to take the sparsity pattern.

matrix The same as x with data changed for ones.

theano.sparse.basic.**sp_sum**(x, *axis=None*, *sparse_grad=False*)

Calculate the sum of a sparse matrix along the specified axis.

It operates a reduction along the specified axis. When *axis* is *None*, it is applied along all axes.

x Sparse matrix.

axis Axis along which the sum is applied. Integer or *None*.

sparse_grad [bool] *True* to have a structured grad.

object The sum of x in a dense format.

The grad implementation is controlled with the *sparse_grad* parameter. *True* will provide a structured grad and *False* will provide a regular grad. For both choices, the grad returns a sparse matrix having the same format as x.

This op does not return a sparse matrix, but a dense tensor matrix.

theano.sparse.basic.**sp_zeros_like**(x)

Construct a sparse matrix of zeros.

x Sparse matrix to take the shape.

matrix The same as x with zero entries for all element.

theano.sparse.basic.**structured_dot**(x, y)

Structured Dot is like dot, except that only the gradient wrt non-zero elements of the sparse matrix a are calculated and propagated.

The output is presumed to be a dense matrix, and is represented by a TensorType instance.

a A sparse matrix.

b A sparse or dense matrix.

matrix The dot product of *a* and *b*.

The grad implemented is structured.

theano.sparse.basic.**sub**(*x*, *y*)

Subtract two matrices, at least one of which is sparse.

This method will provide the right op according to the inputs.

x A matrix variable.

y A matrix variable.

matrix *x* - *y*

At least one of *x* and *y* must be a sparse matrix.

The grad will be structured only when one of the variable will be a dense matrix.

theano.sparse.basic.**true_dot**(*x*, *y*, *grad_preserves_dense=True*)

Operation for efficiently calculating the dot product when one or all operands are sparse. Supported formats are CSC and CSR. The output of the operation is sparse.

x Sparse matrix.

y Sparse matrix or 2d tensor variable.

grad_preserves_dense [bool] If True (default), makes the grad of dense inputs dense. Otherwise the grad is always sparse.

The dot product *x*.'y' in a sparse format.

The grad implemented is regular, i.e. not structured.

theano.sparse.basic.**verify_grad_sparse**(*op*, *pt*, *structured=False*, **args*, ***kwargs*)

Wrapper for theano.test.unittest_tools.py:verify_grad wich converts sparse variables back and forth.

op Op to check.

pt List of inputs to realize the tests.

structured True to tests with a structured grad, False otherwise.

args Other *verify_grad* parameters if any.

kwargs Other *verify_grad* keywords if any.

None

theano.sparse.basic.**vstack**(*blocks*, *format=None*, *dtype=None*)

Stack sparse matrices vertically (row wise).

This wrap the method vstack from scipy.

blocks List of sparse array of compatible shape.

format String representing the output format. Default is csc.

dtype Output dtype.

array The concatenation of the sparse array row wise.

The number of column of the sparse matrix must agree.

The grad implemented is regular, i.e. not structured.

`sparse.sandbox` – Sparse Op Sandbox

API

Convolution-like operations with sparse matrix multiplication.

To read about different sparse formats, see U{http://www-users.cs.umn.edu/~saad/software/SPARSKIT/paper.ps}.

@todo: Automatic methods for determining best sparse format?

class `theano.sparse.sandbox.sp.`**`ConvolutionIndices`**(*use_c_code='/usr/bin/g++'*)

Build indices for a sparse CSC matrix that could implement A (convolve) B.

This generates a sparse matrix M, which generates a stack of image patches when computing the dot product of M with image patch. Convolution is then simply the dot product of (img x M) and the kernels.

static `evaluate`(*inshp*, *kshp*, *strides=(1, 1)*, *nkern=1*, *mode='valid'*, *ws=True*)

Build a sparse matrix which can be used for performing... * convolution: in this case, the dot product of this matrix with the input images will generate a stack of images patches. Convolution is then a tensordot operation of the filters and the patch stack. * sparse local connections: in this case, the sparse matrix allows us to operate the weight matrix as if it were fully-connected. The structured-dot with the input image gives the output for the following layer.

Parameters

- **ker_shape** – shape of kernel to apply (smaller than image)
- **img_shape** – shape of input images
- **mode** – 'valid' generates output only when kernel and image overlap overlap fully. Convolution obtained by zero-padding the input
- **ws** – True if weight sharing, false otherwise
- **(dx,dy)** – offset parameter. In the case of no weight sharing, gives the pixel offset between two receptive fields. With weight sharing gives the offset between the top-left pixels of the generated patches

Return type tuple(indices, indptr, logical_shape, sp_type, out_img_shp)

Returns the structure of a sparse matrix, and the logical dimensions of the image which will be the result of filtering.

theano.sparse.sandbox.sp.**applySparseFilter**(*kerns*, *kshp*, *nkern*, *images*, *imgshp*, *step=(1, 1)*, *bias=None*, *mode='valid'*)

"images" is assumed to be a matrix of shape batch_size x img_size, where the second dimension represents each image in raster order

Output feature map will have shape:

```
batch_size x number of kernels * output_size
```

Note: IMPORTANT: note that this means that each feature map is contiguous in memory.

The memory layout will therefore be: [<feature_map_0> <feature_map_1> ... <feature_map_n>], where <feature_map> represents a "feature map" in raster order

---

Note that the concept of feature map doesn't really apply to sparse filters without weight sharing. Basically, nkern=1 will generate one output img/feature map, nkern=2 a second feature map, etc.

kerns is a 1D tensor, and assume to be of shape:

```
nkern * N.prod(outshp) x N.prod(kshp)
```

Each filter is applied seperately to consecutive output pixels.

Parameters

- **kerns** – nkern*outsize*ksize vector containing kernels
- **kshp** – tuple containing actual dimensions of kernel (not symbolic)
- **nkern** – number of kernels to apply at each pixel in the input image. nkern=1 will apply a single unique filter for each input pixel.
- **images** – bsize x imgsize matrix containing images on which to apply filters
- **imgshp** – tuple containing actual image dimensions (not symbolic)
- **step** – determines number of pixels between adjacent receptive fields (tuple containing dx,dy values)
- **mode** – 'full', 'valid' see CSM.evaluate function for details

Returns out1, symbolic result

Returns out2, logical shape of the output img (nkern,height,width) (after dot product, not of the sparse matrix!)

theano.sparse.sandbox.sp.**convolve**(*kerns*, *kshp*, *nkern*, *images*, *imgshp*, *step=(1, 1)*, *bias=None*, *mode='valid'*, *flatten=True*)

Convolution implementation by sparse matrix multiplication.

Note For best speed, put the matrix which you expect to be smaller as the 'kernel' argument

"images" is assumed to be a matrix of shape batch_size x img_size, where the second dimension represents each image in raster order

If flatten is "False", the output feature map will have shape:

```
batch_size x number of kernels x output_size
```

If flatten is "True", the output feature map will have shape:

```
batch_size x number of kernels * output_size
```

Note: IMPORTANT: note that this means that each feature map (image generate by each kernel) is contiguous in memory. The memory layout will therefore be: [<feature_map_0> <feature_map_1> ... <feature_map_n>], where <feature_map> represents a "feature map" in raster order

---

kerns is a 2D tensor of shape nkern x N.prod(kshp)

Parameters

- **kerns** – 2D tensor containing kernels which are applied at every pixel
- **kshp** – tuple containing actual dimensions of kernel (not symbolic)
- **nkern** – number of kernels/filters to apply. nkern=1 will apply one common filter to all input pixels
- **images** – tensor containing images on which to apply convolution
- **imgshp** – tuple containing image dimensions
- **step** – determines number of pixels between adjacent receptive fields (tuple containing dx,dy values)
- **mode** – 'full', 'valid' see CSM.evaluate function for details
- **sumdims** – dimensions over which to sum for the tensordot operation. By default ((2,),(1,)) assumes kerns is a nkern x kernsize matrix and images is a batchsize x imgsize matrix containing flattened images in raster order
- **flatten** – flatten the last 2 dimensions of the output. By default, instead of generating a batchsize x outsize x nkern tensor, will flatten to batchsize x outsize*nkern

Returns out1, symbolic result

Returns out2, logical shape of the output img (nkern,heigt,width)

Todo test for 1D and think of how to do n-d convolutions

`theano.sparse.sandbox.sp.`**`max_pool`**(*images*, *imgshp*, *maxpoolshp*)

Implements a max pooling layer

Takes as input a 2D tensor of shape batch_size x img_size and performs max pooling. Max pooling downsamples by taking the max value in a given area, here defined by maxpoolshp. Outputs a 2D tensor of shape batch_size x output_size.

Parameters

- **images** – 2D tensor containing images on which to apply convolution. Assumed to be of shape batch_size x img_size

- **imgshp** – tuple containing image dimensions
- **maxpoolshp** – tuple containing shape of area to max pool over

Returns out1, symbolic result (2D tensor)

Returns out2, logical shape of the output

class theano.sparse.sandbox.sp2.**Binomial**(*format*, *dtype*)

Return a sparse matrix having random values from a binomial density having number of experiment *n* and probability of succes *p*.

WARNING: This Op is NOT deterministic, as calling it twice with the same inputs will NOT give the same result. This is a violation of Theano's contract for Ops

Parameters

- **n** – Tensor scalar representing the number of experiment.
- **p** – Tensor scalar representing the probability of success.
- **shape** – Tensor vector for the output shape.

Returns A sparse matrix of integers representing the number of success.

class theano.sparse.sandbox.sp2.**Multinomial**(*use_c_code='/usr/bin/g++'*)

Return a sparse matrix having random values from a multinomial density having number of experiment *n* and probability of succes *p*.

WARNING: This Op is NOT deterministic, as calling it twice with the same inputs will NOT give the same result. This is a violation of Theano's contract for Ops

Parameters

- **n** – Tensor type vector or scalar representing the number of experiment for each row. If *n* is a scalar, it will be used for each row.
- **p** – Sparse matrix of probability where each row is a probability vector representing the probability of succes. N.B. Each row must sum to one.

Returns A sparse matrix of random integers from a multinomial density for each row.

Note It will works only if *p* have csr format.

class theano.sparse.sandbox.sp2.**Poisson**(*use_c_code='/usr/bin/g++'*)

Return a sparse having random values from a Poisson density with mean from the input.

WARNING: This Op is NOT deterministic, as calling it twice with the same inputs will NOT give the same result. This is a violation of Theano's contract for Ops

Parameters **x** – Sparse matrix.

Returns A sparse matrix of random integers of a Poisson density with mean of *x* element wise.

scalar – Symbolic Scalar Types, Ops [doc TODO]

gof – Theano Internals [doc TODO]

fgraph – Graph Container [doc TODO]

Guide

FunctionGraph

FunctionGraph Features

FunctionGraph Feature List

- ReplaceValidate
- DestroyHandler

Reference

class fgraph.**FunctionGraph**

TODO

Note: FunctionGraph(inputs, outputs) clones the inputs by default. To avoid this behavior, add the parameter clone=False. This is needed as we do not want cached constants in fgraph.

---

toolbox – [doc TODO]

Guide

class theano.gof.toolbox.**Bookkeeper**(*object*)

class theano.gof.toolbox.**History**(*object*)

revert(*fgraph*, *checkpoint*)

Reverts the graph to whatever it was at the provided checkpoint (undoes all replacements). A checkpoint at any given time can be obtained using self.checkpoint().

class theano.gof.toolbox.**Validator**(*object*)

class theano.gof.toolbox.**ReplaceValidate**(*History*, *Validator*)

replace_validate(*fgraph*, *var*, *new_var*, *reason=None*)

class theano.gof.toolbox.**NodeFinder**(*Bookkeeper*)

class theano.gof.toolbox.**PrintListener**(*object*)

`type` – Interface for types of variables

Reference WRITEME Defines the *Type* class.

class `theano.gof.type.`**`CDataType`**(*ctype*, *freefunc=None*)
Represents opaque C data to be passed around. The intent is to ease passing arbitrary data between ops C code.

class `theano.gof.type.`**`CLinkerType`**
Interface specification for Types that can be arguments to a *CLinkerOp*.

A CLinkerType instance is mainly reponsible for providing the C code that interfaces python objects with a C *CLinkerOp* implementation.

See WRITEME for a general overview of code generation by *CLinker*.

`c_cleanup`(*name*, *sub*)
Return c code to clean up after *c_extract*.

This returns C code that should deallocate whatever *c_extract* allocated or decrease the reference counts. Do not decrease py_%(name)s's reference count.

WRITEME

Parameters

- ***name*: WRITEME** WRITEME
- ***sub*: WRITEME** WRITEME

Exceptions

- *MethodNotDefined*: Subclass does not implement this method

`c_code_cache_version`()
Return a tuple of integers indicating the version of this Type.

An empty tuple indicates an 'unversioned' Type that will not be cached between processes.

The cache mechanism may erase cached modules that have been superceded by newer versions. See *ModuleCache* for details.

`c_declare`(*name*, *sub*, *check_input=True*)
Required: Return c code to declare variables that will be instantiated by *c_extract*.

Example: .. code-block: python

return "PyObject ** addr_of_%(name)s;"

Parameters

- **name** (*string*) – the name of the `PyObject *` pointer that will the value for this Type
- **sub** (*dict string -> string*) – a dictionary of special codes. Most importantly sub['fail']. See CLinker for more info on *sub* and `fail`.

Note It is important to include the *name* inside of variables which are declared here, so that name collisions do not occur in the source file that is generated.

Note The variable called `name` is not necessarily defined yet where this code is inserted. This code might be inserted to create class variables for example, whereas the variable `name` might only exist inside certain functions in that class.

Todo Why should variable declaration fail? Is it even allowed to?

Exceptions

- *MethodNotDefined*: Subclass does not implement this method

c_extract(*name*, *sub*, *check_input=True*)

Required: Return c code to extract a PyObject * instance.

The code returned from this function must be templated using `%(name)s`, representing the name that the caller wants to call this *Variable*. The Python object self.data is in a variable called "py_%(name)s" and this code must set the variables declared by c_declare to something representative of py_%(name)s. If the data is improper, set an appropriate exception and insert "%(fail)s".

Todo Point out that template filling (via sub) is now performed by this function. –jpt

Example: .. code-block: python

return "if (py_%(name)s == Py_None)" + addr_of_%(name)s = &py_%(name)s;" + "else" + { PyErr_SetString(PyExc_ValueError, 'was expecting None'); %(fail)s;}"

Parameters

- **name** (*string*) – the name of the `PyObject *` pointer that will store the value for this Type
- **sub** (*dict string -> string*) – a dictionary of special codes. Most importantly sub['fail']. See CLinker for more info on *sub* and `fail`.

Exceptions

- *MethodNotDefined*: Subclass does not implement this method

c_extract_out(*name*, *sub*, *check_input=True*)

Optional: C code to extract a PyObject * instance.

Unlike c_extract, c_extract_out has to accept Py_None, meaning that the variable should be left uninitialized.

c_init(*name*, *sub*)

Required: Return c code to initialize the variables that were declared by self.c_declare()

Example: .. code-block: python

return "addr_of_%(name)s = NULL;"

Note The variable called `name` is not necessarily defined yet where this code is inserted. This code might be inserted in a class constructor for example, whereas the variable `name` might only exist inside certain functions in that class.

Todo Why should variable initialization fail? Is it even allowed to?

c_is_simple()
Optional: Return True for small or builtin C types.

A hint to tell the compiler that this type is a builtin C type or a small struct and that its memory footprint is negligible. Simple objects may be passed on the stack.

c_literal(*data*)
Optional: WRITEME

Parameters

- ***data*: WRITEME** WRITEME

Exceptions

- *MethodNotDefined*: Subclass does not implement this method

c_sync(*name*, *sub*)
Required: Return c code to pack C types back into a PyObject.

The code returned from this function must be templated using "%(name)s", representing the name that the caller wants to call this Variable. The returned code may set "py_%(name)s" to a PyObject* and that PyObject* will be accessible from Python via variable.data. Do not forget to adjust reference counts if "py_%(name)s" is changed from its original value.

Parameters

- ***name*: WRITEME** WRITEME
- ***sub*: WRITEME** WRITEME

Exceptions

- *MethodNotDefined*: Subclass does not implement this method

class theano.gof.type.**Generic**
Represents a generic Python object.

This class implements the *PureType* and *CLinkerType* interfaces for generic PyObject instances.

EXAMPLE of what this means, or when you would use this type.

WRITEME

class theano.gof.type.**PureType**
Interface specification for variable type instances.

A *Type* instance is mainly reponsible for two things:

•creating *Variable* instances (conventionally, *__call__* does this), and

•filtering a value assigned to a *Variable* so that the value conforms to restrictions imposed by the type (also known as casting, this is done by *filter*),

class **Constant**(*type*, *data*, *name=None*)

A *Constant* is a *Variable* with a *value* field that cannot be changed at runtime.

Constant nodes make eligible numerous optimizations: constant inlining in C code, constant folding, etc.

clone()

We clone this object, but we don't clone the data to lower memory requirement We suppose that the data will never change.

value

read-only data access method

class PureType.**Variable**(*type*, *owner=None*, *index=None*, *name=None*)

A *Variable* is a node in an expression graph that represents a variable.

The inputs and outputs of every *Apply* (theano.gof.Apply) are *Variable* instances. The input and output arguments to create a *function* are also *Variable* instances. A *Variable* is like a strongly-typed variable in some other languages; each *Variable* contains a reference to a *Type* instance that defines the kind of value the *Variable* can take in a computation.

A *Variable* is a container for four important attributes:

- `type` a *Type* instance defining the kind of value this *Variable* can have,
- `owner` either None (for graph roots) or the *Apply* instance of which *self* is an output,
- `index` the integer such that `owner.outputs[index] is this_variable` (ignored if *owner* is None)
- `name` a string to use in pretty-printing and debugging.

There are a few kinds of Variables to be aware of: A Variable which is the output of a symbolic computation has a reference to the Apply instance to which it belongs (property: owner) and the position of itself in the owner's output list (property: index).

- *Variable* (this base type) is typically the output of a symbolic computation,
- *Constant* (a subclass) which adds a default and un-replaceable `value`, and requires that owner is None
- *TensorVariable* subclass of Variable that represents a numpy.ndarray object
- *TensorSharedVariable* Shared version of TensorVariable
- *SparseVariable* subclass of Variable that represents a scipy.sparse.{csc,csr}_matrix object
- *CudaNdarrayVariable* subclass of Variable that represents our object on the GPU that is a subset of numpy.ndarray
- *RandomVariable*

A Variable which is the output of a symbolic computation will have an owner not equal to None.

Using the Variables' owner field and the Apply nodes' inputs fields, one can navigate a graph from an output all the way to the inputs. The opposite direction is not possible until an FunctionGraph has annotated the Variables with the clients field, ie, before the compilation process has begun a Variable does not know which Apply nodes take it as input.

Code Example

```
import theano
from theano import tensor

a = tensor.constant(1.5)        # declare a symbolic constant
b = tensor.fscalar()            # declare a symbolic floating-point scalar

c = a + b                       # create a simple expression

f = theano.function([b], [c])   # this works because a has a value associated with

assert 4.0 == f(2.5)            # bind 2.5 to an internal copy of b and evaluate a

theano.function([a], [c])       # compilation error because b (required by c) is u

theano.function([a,b], [c])     # compilation error because a is constant, it can'

d = tensor.value(1.5)           # create a value similar to the constant 'a'
e = d + b
theano.function([d,b], [e])     # this works.  d's default value of 1.5 is ignored
```

The python variables `a,b,c` all refer to instances of type *Variable*. The *Variable* refered to by *a* is also an instance of *Constant*.

compile.function uses each *Apply* instance's *inputs* attribute together with each Variable's *owner* field to determine which inputs are necessary to compute the function's outputs.

clone()
Return a new Variable like self.
Return type Variable instance
Returns a new Variable instance (or subclass instance) with no owner or index.
Note tags are copied to the returned instance.
Note name is copied to the returned instance.

eval(*inputs_to_values=None*)
Evaluates this variable.

inputs_to_values: a dictionary mapping theano Variables to values.

PureType.**convert_variable**(*var*)
Patch variable so that its type will match self, if possible.

If the variable can't be converted, this should return None.

The conversion can only happen if the following implication is true for all possible *val*.

self.is_valid_value(val) => var.type.is_valid_value(val)

For the majority of types this means that you can only have non-broadcastable dimensions become broadcastable and not the inverse.

The default is to not convert anything which is always safe.

PureType.**filter**(*data*, *strict=False*, *allow_downcast=None*)
Required: Return data or an appropriately wrapped/converted data.

Subclass implementation should raise a TypeError exception if the data is not of an acceptable type.

If strict is True, the data returned must be the same as the data passed as an argument. If it is False, and allow_downcast is True, filter may cast it to an appropriate type. If allow_downcast is False, filter may only upcast it, not lose precision. If allow_downcast is None (default), the behaviour can be Type-dependent, but for now it means only Python floats can be downcasted, and only to floatX scalars.

Exceptions

- *MethodNotDefined*: subclass doesn't implement this function.

`PureType.`**`filter_variable`**`(`*other*, *allow_convert=True*`)`
Convert a symbolic variable into this Type, if compatible.

For the moment, the only Types compatible with one another are TensorType and CudaNdarrayType, provided they have the same number of dimensions, same broadcasting pattern, and same dtype.

If Types are not compatible, a TypeError should be raised.

`PureType.`**`is_valid_value`**`(`*a*`)`
Required: Return True for any python object *a* that would be a legal value for a Variable of this Type

`PureType.`**`make_variable`**`(`*name=None*`)`
Return a new *Variable* instance of Type *self*.

Parameters

- ***name*: None or str** A pretty string for printing and debugging.

`PureType.`**`value_validity_msg`**`(`*a*`)`
Optional: return a message explaining the output of is_valid_value

`PureType.`**`values_eq`**`(`*a*, *b*`)`
Return True if a and b can be considered exactly equal.

a and b are assumed to be valid values of this Type.

`PureType.`**`values_eq_approx`**`(`*a*, *b*`)`
Return True if a and b can be considered approximately equal.

Parameters

- **a** – a potential value for a Variable of this Type.
- **b** – a potential value for a Variable of this Type.

Return type Bool

This function is used by theano debugging tools to decide whether two values are equivalent, admitting a certain amount of numerical instability. For example, for floating-point numbers this function should be an approximate comparison.

By default, this does an exact comparison.

class theano.gof.type.**SingletonType**
Convenient Base class for a Type subclass with no attributes

It saves having to implement __eq__ and __hash__

class theano.gof.type.**Type**
Convenience wrapper combining *PureType* and *CLinkerType*.

Theano comes with several subclasses of such as:

•*Generic*: for any python type

•*TensorType*: for numpy.ndarray

•*SparseType*: for scipy.sparse

But you are encouraged to write your own, as described in WRITEME.

The following following code illustrates the use of a Type instance, here tensor.fvector:

```
# Declare a symbolic floating-point vector using __call__
b = tensor.fvector()

# Create a second Variable with the same Type instance
c = tensor.fvector()
```

Whenever you create a symbolic variable in theano (technically, *Variable*) it will contain a reference to a Type instance. That reference is typically constant during the lifetime of the Variable. Many variables can refer to a single Type instance, as do b and c above. The Type instance defines the kind of value which might end up in that variable when executing a *Function*. In this sense, theano is like a strongly-typed language because the types are included in the graph before the values. In our example above, b is a Variable which is guaranteed to correspond to a numpy.ndarray of rank 1 when we try to do some computations with it.

Many *Op* instances will raise an exception if they are applied to inputs with incorrect types. Type references are also useful to do type-checking in pattern-based optimizations.

utils – Utilities functions operating on the graph

Reference

exception theano.gof.utils.**MethodNotDefined**
To be raised by functions defined as part of an interface.

When the user sees such an error, it is because an important interface function has been left out of an implementation class.

theano.gof.utils.**add_tag_trace**(*thing*, *user_line=1*)
Add tag.trace to an node or variable.

The argument is returned after being affected (inplace). :param thing: the object where we add .tag.trace :param user_line: The max number of user line to keep.

Note we alse use config.traceback.limit for the maximum number of stack level we look.

theano.gof.utils.**deprecated**(*filename*, *msg=''*)
Decorator which will print a warning message on the first call.

Use it like this:

```
@deprecated('myfile', 'do something different...')
def fn_name(...)
    ...
```

And it will print:

```
WARNING myfile.fn_name deprecated. do something different...
```

theano.gof.utils.**difference**(*seq1*, *seq2*)
Returns all elements in seq1 which are not in seq2: i.e `seq1\seq2`

theano.gof.utils.**flatten**(*a*)
Recursively flatten tuple, list and set in a list.

theano.gof.utils.**give_variables_names**(*variables*)
Gives unique names to an iterable of variables. Modifies input.

This function is idempotent.

theano.gof.utils.**hash_from_dict**(*d*)
Work around the fact that dict are not hashable in python

This request that all object have a sorted order that depend only on the key of the object. We support only integer/float/string keys.

Also, we transform values that are list into tuple as list are not hashable.

Note special case for OrderedDict, it use the order of the dict, so the key don't need to be sortable.

theano.gof.utils.**hash_from_file**(*file_path*)
Return the MD5 hash of a file.

theano.gof.utils.**memoize**(*f*)
Cache the return value for each tuple of arguments (which must be hashable)

theano.gof.utils.**remove**(*predicate*, *coll*)
Return those items of collection for which predicate(item) is true.

```
>>> from itertoolz import remove
>>> def even(x):
...     return x % 2 == 0
>>> remove(even, [1, 2, 3, 4])
[1, 3]
```

theano.gof.utils.**simple_extract_stack**(*f=None*, *limit=None*)
This is traceback.extract_stack from python 2.7 with this change:

•Comment the update of the cache

This is because this update cause an call to os.stat to get the line content. This cause too much long on cluster.

theano.gof.utils.**toposort**(*prereqs_d*)
Sorts prereqs_d.keys() topologically.

prereqs_d[x] contains all the elements that must come before x in the ordering.

theano.gof.utils.**uniq**(*seq*)
Do not use set, this must always return the same value at the same index. If we just exchange other values, but keep the same pattern of duplication, we must keep the same order.

misc.pkl_utils - Tools for serialization.

theano.misc.pkl_utils.**dump**(*obj*, *file_handler*, *protocol=2*, *persistent_id=<class 'theano.misc.pkl_utils.PersistentSharedVariableID'>*)
Pickles an object to a zip file using external persistence.

Parameters

- **obj** (*object*) – The object to pickle.
- **file_handler** (*file*) – The file handle to save the object to.
- **protocol** (*int, optional*) – The pickling protocol to use. Unlike Python's built-in pickle, the default is set to *2* instead of 0 for Python 2. The Python 3 default (level 3) is maintained.
- **persistent_id** (*callable*) – The callable that persists certain objects in the object hierarchy to separate files inside of the zip file. For example, `PersistentNdarrayID` saves any `numpy.ndarray` to a separate NPY file inside of the zip file.

Note: The final file is simply a zipped file containing at least one file, *pkl*, which contains the pickled object. It can contain any other number of external objects. Note that the zip files are compatible with NumPy's `numpy.load()` function.

```
>>> import theano
>>> foo_1 = theano.shared(0, name='foo')
>>> foo_2 = theano.shared(1, name='foo')
>>> with open('model.zip', 'w') as f:
...     dump((foo_1, foo_2, numpy.array(2)), f)
>>> numpy.load('model.zip').keys()
['foo', 'foo_2', 'array_0', 'pkl']
>>> numpy.load('model.zip')['foo']
array(0)
>>> with open('model.zip') as f:
...     foo_1, foo_2, array = load(f)
>>> array
array(2)
```

theano.misc.pkl_utils.**load**(*f*, *persistent_load=<class 'theano.misc.pkl_utils.PersistentNdarrayLoad'>*)
Load a file that was dumped to a zip file.

Parameters

- **f** (*file*) – The file handle to the zip file to load the object from.
- **persistent_load** (*callable, optional*) – The persistent loading function to use for unpickling. This must be compatible with the *persisten_id* function used when pickling.

See also:

Loading and Saving

`scan` – Looping in Theano

Guide

The scan functions provides the basic functionality needed to do loops in Theano. Scan comes with many whistles and bells, which we will introduce by way of examples.

Simple loop with accumulation: Computing A^k Assume that, given k you want to get `A**k` using a loop. More precisely, if A is a tensor you want to compute `A**k` elemwise. The python/numpy code might look like:

```
result = 1
for i in xrange(k):
  result = result * A
```

There are three things here that we need to handle: the initial value assigned to `result`, the accumulation of results in `result`, and the unchanging variable `A`. Unchanging variables are passed to scan as `non_sequences`. Initialization occurs in `outputs_info`, and the accumulation happens automatically.

The equivalent Theano code would be:

```
k = T.iscalar("k")
A = T.vector("A")

# Symbolic description of the result
result, updates = theano.scan(fn=lambda prior_result, A: prior_result * A,
                              outputs_info=T.ones_like(A),
                              non_sequences=A,
                              n_steps=k)

# We only care about A**k, but scan has provided us with A**1 through A**k.
# Discard the values that we don't care about. Scan is smart enough to
# notice this and not waste memory saving them.
final_result = result[-1]

# compiled function that returns A**k
power = theano.function(inputs=[A,k], outputs=final_result, updates=updates)

print power(range(10),2)
print power(range(10),4)
```

Let us go through the example line by line. What we did is first to construct a function (using a lambda expression) that given `prior_result` and `A` returns `prior_result * A`. The order of parameters is fixed by scan: the output of the prior call to `fn` (or the initial value, initially) is the first parameter, followed by all non-sequences.

Next we initialize the output as a tensor with same shape and dtype as `A`, filled with ones. We give `A` to scan as a non sequence parameter and specify the number of steps `k` to iterate over our lambda expression.

Scan returns a tuple containing our result (`result`) and a dictionary of updates (empty in this case). Note that the result is not a matrix, but a 3D tensor containing the value of `A**k` for each step. We want the last value (after `k` steps) so we compile a function to return just that. Note that there is an optimization, that at compile time will detect that you are using just the last value of the result and ensure that scan does not store all the intermediate values that are used. So do not worry if `A` and `k` are large.

Iterating over the first dimension of a tensor: Calculating a polynomial In addition to looping a fixed number of times, scan can iterate over the leading dimension of tensors (similar to Python's `for x in a_list`).

The tensor(s) to be looped over should be provided to scan using the `sequence` keyword argument.

Here's an example that builds a symbolic calculation of a polynomial from a list of its coefficients:

```
coefficients = theano.tensor.vector("coefficients")
x = T.scalar("x")

max_coefficients_supported = 10000

# Generate the components of the polynomial
components, updates = theano.scan(fn=lambda coefficient, power, free_variable: coefficient
                                  outputs_info=None,
                                  sequences=[coefficients, theano.tensor.arange(max_coeffi
                                  non_sequences=x)
# Sum them up
polynomial = components.sum()

# Compile a function
calculate_polynomial = theano.function(inputs=[coefficients, x], outputs=polynomial)

# Test
test_coefficients = numpy.asarray([1, 0, 2], dtype=numpy.float32)
test_value = 3
print calculate_polynomial(test_coefficients, test_value)
print 1.0 * (3 ** 0) + 0.0 * (3 ** 1) + 2.0 * (3 ** 2)
```

There are a few things to note here.

First, we calculate the polynomial by first generating each of the coefficients, and then summing them at the end. (We could also have accumulated them along the way, and then taken the last one, which would have been more memory-efficient, but this is an example.)

Second, there is no accumulation of results, we can set `outputs_info` to `None`. This indicates to scan that it doesn't need to pass the prior result to `fn`.

The general order of function parameters to `fn` is:

```
sequences (if any), prior result(s) (if needed), non-sequences (if any)
```

Third, there's a handy trick used to simulate python's `enumerate`: simply include `theano.tensor.arange` to the sequences.

Fourth, given multiple sequences of uneven lengths, scan will truncate to the shortest of them. This makes it safe to pass a very long arange, which we need to do for generality, since arange must have its length specified at creation time.

Simple accumulation into a scalar, ditching lambda Although this example would seem almost self-explanatory, it stresses a pitfall to be careful of: the initial output state that is supplied, that is `outputs_info`, must be of a **shape similar to that of the output variable** generated at each iteration and moreover, it **must not involve an implicit downcast** of the latter.

```
import numpy as np
import theano
import theano.tensor as T

up_to = T.iscalar("up_to")

# define a named function, rather than using lambda
def accumulate_by_adding(arange_val, sum_to_date):
    return sum_to_date + arange_val
seq = T.arange(up_to)

# An unauthorized implicit downcast from the dtype of 'seq', to that of
# 'T.as_tensor_variable(0)' which is of dtype 'int8' by default would occur
# if this instruction were to be used instead of the next one:
# outputs_info = T.as_tensor_variable(0)

outputs_info = T.as_tensor_variable(np.asarray(0, seq.dtype))
scan_result, scan_updates = theano.scan(fn=accumulate_by_adding,
                                        outputs_info=outputs_info,
                                        sequences=seq)
triangular_sequence = theano.function(inputs=[up_to], outputs=scan_result)

# test
some_num = 15
print triangular_sequence(some_num)
print [n * (n + 1) // 2 for n in xrange(some_num)]
```

Another simple example Unlike some of the prior examples, this one is hard to reproduce except by using scan.

This takes a sequence of array indices, and values to place there, and a "model" output array (whose shape and dtype will be mimicked), and produces a sequence of arrays with the shape and dtype of the model, with all values set to zero except at the provided array indices.

```
location = T.imatrix("location")
values = T.vector("values")
output_model = T.matrix("output_model")
```

```
def set_value_at_position(a_location, a_value, output_model):
    zeros = T.zeros_like(output_model)
    zeros_subtensor = zeros[a_location[0], a_location[1]]
    return T.set_subtensor(zeros_subtensor, a_value)

result, updates = theano.scan(fn=set_value_at_position,
                              outputs_info=None,
                              sequences=[location, values],
                              non_sequences=output_model)

assign_values_at_positions = theano.function(inputs=[location, values, output_model], outpu

# test
test_locations = numpy.asarray([[1, 1], [2, 3]], dtype=numpy.int32)
test_values = numpy.asarray([42, 50], dtype=numpy.float32)
test_output_model = numpy.zeros((5, 5), dtype=numpy.float32)
print assign_values_at_positions(test_locations, test_values, test_output_model)
```

This demonstrates that you can introduce new Theano variables into a scan function.

Using shared variables - Gibbs sampling Another useful feature of scan, is that it can handle shared variables. For example, if we want to implement a Gibbs chain of length 10 we would do the following:

```
W = theano.shared(W_values) # we assume that ''W_values'' contains the
                            # initial values of your weight matrix

bvis = theano.shared(bvis_values)
bhid = theano.shared(bhid_values)

trng = T.shared_randomstreams.RandomStreams(1234)

def OneStep(vsample) :
    hmean = T.nnet.sigmoid(theano.dot(vsample, W) + bhid)
    hsample = trng.binomial(size=hmean.shape, n=1, p=hmean)
    vmean = T.nnet.sigmoid(theano.dot(hsample, W.T) + bvis)
    return trng.binomial(size=vsample.shape, n=1, p=vmean,
                         dtype=theano.config.floatX)

sample = theano.tensor.vector()

values, updates = theano.scan(OneStep, outputs_info=sample, n_steps=10)

gibbs10 = theano.function([sample], values[-1], updates=updates)
```

The first, and probably most crucial observation is that the updates dictionary becomes important in this case. It links a shared variable with its updated value after k steps. In this case it tells how the random streams get updated after 10 iterations. If you do not pass this update dictionary to your function, you will always get the same 10 sets of random numbers. You can even use the `updates` dictionary afterwards. Look at this example :

```
a = theano.shared(1)
values, updates = theano.scan(lambda: {a: a+1}, n_steps=10)
```

In this case the lambda expression does not require any input parameters and returns an update dictionary which tells how `a` should be updated after each step of scan. If we write :

```
b = a + 1
c = updates[a] + 1
f = theano.function([], [b, c], updates=updates)

print b
print c
print a.value
```

We will see that because `b` does not use the updated version of `a`, it will be 2, `c` will be 12, while `a.value` is `11`. If we call the function again, `b` will become 12, `c` will be 22 and `a.value` 21. If we do not pass the `updates` dictionary to the function, then `a.value` will always remain 1, `b` will always be 2 and `c` will always be `12`.

The second observation is that if we use shared variables (`W`, `bvis`, `bhid`) but we do not iterate over them (ie scan doesn't really need to know anything in particular about them, just that they are used inside the function applied at each step) you do not need to pass them as arguments. Scan will find them on its own and add them to the graph. However, passing them to the scan function is a good practice, as it avoids Scan Op calling any earlier (external) Op over and over. This results in a simpler computational graph, which speeds up the optimization and the execution. To pass the shared variables to Scan you need to put them in a list and give it to the `non_sequences` argument. Here is the Gibbs sampling code updated:

```
W = theano.shared(W_values) # we assume that ''W_values'' contains the
                            # initial values of your weight matrix

bvis = theano.shared(bvis_values)
bhid = theano.shared(bhid_values)

trng = T.shared_randomstreams.RandomStreams(1234)

# OneStep, with explicit use of the shared variables (W, bvis, bhid)
def OneStep(vsample, W, bvis, bhid):
    hmean = T.nnet.sigmoid(theano.dot(vsample, W) + bhid)
    hsample = trng.binomial(size=hmean.shape, n=1, p=hmean)
    vmean = T.nnet.sigmoid(theano.dot(hsample, W.T) + bvis)
    return trng.binomial(size=vsample.shape, n=1, p=vmean,
                         dtype=theano.config.floatX)

sample = theano.tensor.vector()

# The new scan, with the shared variables passed as non_sequences
values, updates = theano.scan(fn=OneStep,
                              outputs_info=sample,
                              non_sequences=[W, bvis, bhid],
                              n_steps=10)

gibbs10 = theano.function([sample], values[-1], updates=updates)
```

Using shared variables - the strict flag As we just saw, passing the shared variables to scan may result in a simpler computational graph, which speeds up the optimization and the execution. A good way to remember to pass every shared variable used during scan is to use the `strict` flag. When set to true, scan assumes that all the necessary shared variables in `fn` are passed as a part of `non_sequences`. This has to be ensured by the user. Otherwise, it will result in an error.

Using the previous Gibbs sampling example:

```
# The new scan, using strict=True
values, updates = theano.scan(fn=OneStep,
                              outputs_info=sample,
                              non_sequences=[W, bvis, bhid],
                              n_steps=10,
                              strict=True)
```

If you omit to pass `W`, `bvis` or `bhid` as a `non_sequence`, it will result in an error.

Multiple outputs, several taps values - Recurrent Neural Network with Scan The examples above showed simple uses of scan. However, scan also supports referring not only to the prior result and the current sequence value, but also looking back more than one step.

This is needed, for example, to implement a RNN using scan. Assume that our RNN is defined as follows :

$$x(n) = \tanh(Wx(n-1) + W_1^{in}u(n) + W_2^{in}u(n-4) + W^{feedback}y(n-1))$$
$$y(n) = W^{out}x(n-3)$$

Note that this network is far from a classical recurrent neural network and might be useless. The reason we defined as such is to better illustrate the features of scan.

In this case we have a sequence over which we need to iterate `u`, and two outputs `x` and `y`. To implement this with scan we first construct a function that computes one iteration step :

```
def oneStep(u_tm4, u_t, x_tm3, x_tm1, y_tm1, W, W_in_1, W_in_2,  W_feedback, W_out):

  x_t = T.tanh(theano.dot(x_tm1, W) + \
               theano.dot(u_t,   W_in_1) + \
               theano.dot(u_tm4, W_in_2) + \
               theano.dot(y_tm1, W_feedback))
  y_t = theano.dot(x_tm3, W_out)

  return [x_t, y_t]
```

As naming convention for the variables we used `a_tmb` to mean `a` at `t-b` and `a_tpb` to be `a` at `t+b`. Note the order in which the parameters are given, and in which the result is returned. Try to respect chronological order among the taps (time slices of sequences or outputs) used. For scan is crucial only for the variables representing the different time taps to be in the same order as the one in which these taps are given. Also, not only taps should respect an order, but also variables, since this is how scan figures out what should be represented by what. Given that we have all the Theano variables needed we construct our RNN as follows :

```
u  = T.matrix() # it is a sequence of vectors
x0 = T.matrix() # initial state of x has to be a matrix, since
```

```
                # it has to cover x[-3]
y0 = T.vector() # y0 is just a vector since scan has only to provide
                # y[-1]


([x_vals, y_vals], updates) = theano.scan(fn=oneStep,
                                          sequences=dict(input=u, taps=[-4,-0]),
                                          outputs_info=[dict(initial=x0, taps=[-3,-1]), y0
                                          non_sequences=[W, W_in_1, W_in_2, W_feedback, W_
                                          strict=True)
     # for second input y, scan adds -1 in output_taps by default
```

Now `x_vals` and `y_vals` are symbolic variables pointing to the sequence of x and y values generated by iterating over u. The `sequence_taps`, `outputs_taps` give to scan information about what slices are exactly needed. Note that if we want to use `x[t-k]` we do not need to also have `x[t-(k-1)]`, `x[t-(k-2)]`, .., but when applying the compiled function, the numpy array given to represent this sequence should be large enough to cover this values. Assume that we compile the above function, and we give as `u` the array `uvals = [0,1,2,3,4,5,6,7,8]`. By abusing notations, scan will consider `uvals[0]` as `u[-4]`, and will start scaning from `uvals[4]` towards the end.

Conditional ending of Scan Scan can also be used as a `repeat-until` block. In such a case scan will stop when either the maximal number of iteration is reached, or the provided condition evaluates to True.

For an example, we will compute all powers of two smaller then some provided value `max_value`.

```
def power_of_2(previous_power, max_value):
    return previous_power*2, theano.scan_module.until(previous_power*2 > max_value)

max_value = T.scalar()
values, _ = theano.scan(power_of_2,
                        outputs_info = T.constant(1.),
                        non_sequences = max_value,
                        n_steps = 1024)

f = theano.function([max_value], values)

print f(45)
```

As you can see, in order to terminate on condition, the only thing required is that the inner function `power_of_2` to return also the condition wrapped in the class `theano.scan_module.until`. The condition has to be expressed in terms of the arguments of the inner function (in this case `previous_power` and `max_value`).

As a rule, scan always expects the condition to be the last thing returned by the inner function, otherwise an error will be raised.

Optimizing Scan's performance This section covers some ways to improve performance of a Theano function using Scan.

Minimizing Scan usage Scan makes it possible to define simple and compact graphs that can do the same work as much larger and more complicated graphs. However, it comes with a significant overhead. As such, when performance is the objective, a good rule of thumb is to perform as much of the computation as possible outside of Scan. This may have the effect of increasing memory usage but can also reduce the overhead introduces by using Scan.

Explicitly passing inputs of the inner function to scan It is possible, inside of Scan, to use variables previously defined outside of the Scan without explicitly passing them as inputs to the Scan. However, it is often more efficient to explicitly pass them as non-sequence inputs instead. Section *Using shared variables - Gibbs sampling* provides an explanation for this and section *Using shared variables - the strict flag* describes the *strict* flag, a tool that Scan provides to help ensure that the inputs to the function inside Scan have all been provided as explicit inputs to the `scan()` function.

Deactivating garbage collecting in Scan Deactivating the garbage collection for Scan can allow it to reuse memory between executions instead of always having to allocate new memory. This can improve performance at the cost of increased memory usage. By default, Scan reuses memory between iterations of the same execution but frees the memory after the last iteration.

There are two ways to achieve this, using the Theano flag `config.scan.allow_gc` and setting it to False, or using the argument `allow_gc` of the function theano.scan() and set it to False (when a value is not provided for this argument, the value of the flag `config.scan.allow_gc` is used).

Graph optimizations This one is simple but still worth pointing out. Theano is able to automatically recognize and optimize many computation patterns. However, there are patterns that Theano doesn't optimize because doing so would change the user interface (such as merging shared variables together into a single one, for instance). Additionaly, Theano doesn't catch every case that it could optimize and so it remains useful for performance that the user defines an efficient graph in the first place. This is also the case, and sometimes even more so, for the graph inside of Scan. This is because it will be executed many times for every execution of the Theano function that contains it.

The LSTM tutorial on DeepLearning.net provides an example of an optimization that Theano cannot perform. Instead of performing many matrix multiplications between matrix x_t and each of the shared matrices W_i, W_c, W_f and W_o, the matrices W_*, are merged into a single shared matrix W and the graph performs a single larger matrix multiplication between W and x_t. The resulting matrix is then sliced to obtain the results of that the small individual matrix multiplications would have produced. This optimization replaces several small and inefficient matrix multiplications by a single larger one and thus improves performance at the cost of a potentially higher memory usage.

reference

This module provides the Scan Op.

Scanning is a general form of recurrence, which can be used for looping. The idea is that you *scan* a function along some input sequence, producing an output at each time-step that can be seen (but not modified) by the function at the next time-step. (Technically, the function can see the previous K time-steps of your outputs and L time steps (from the past and future) of your inputs.

So for example, `sum()` could be computed by scanning the `z+x_i` function over a list, given an initial state of `z=0`.

Special cases:

- A *reduce* operation can be performed by returning only the last output of a `scan`.
- A *map* operation can be performed by applying a function that ignores previous steps of the outputs.

Often a for-loop can be expressed as a `scan()` operation, and `scan` is the closest that theano comes to looping. The advantage of using `scan` over for loops is that it allows the number of iterations to be a part of the symbolic graph.

The Scan Op should typically be used by calling any of the following functions: `scan()`, `map()`, `reduce()`, `foldl()`, `foldr()`.

`theano.map`(*fn*, *sequences*, *non_sequences=None*, *truncate_gradient=-1*, *go_backwards=False*, *mode=None*, *name=None*)

Similar behaviour as python's map.

fn The function that `map` applies at each iteration step (see `scan` for more info).

sequences List of sequences over which `map` iterates (see `scan` for more info).

non_sequences List of arguments passed to `fn`. `map` will not iterate over these arguments (see `scan` for more info).

truncate_gradient See `scan`.

go_backwards [bool] Decides the direction of iteration. True means that sequences are parsed from the end towards the begining, while False is the other way around.

mode See `scan`.

name See `scan`.

`theano.reduce`(*fn*, *sequences*, *outputs_info*, *non_sequences=None*, *go_backwards=False*, *mode=None*, *name=None*)

Similar behaviour as python's reduce.

fn The function that `reduce` applies at each iteration step (see `scan` for more info).

sequences List of sequences over which `reduce` iterates (see `scan` for more info).

outputs_info List of dictionaries describing the outputs of reduce (see `scan` for more info).

non_sequences

List of arguments passed to `fn`. `reduce` will not iterate over these arguments (see `scan` for more info).

go_backwards [bool] Decides the direction of iteration. True means that sequences are parsed from the end towards the begining, while False is the other way around.

mode See `scan`.

name See `scan`.

`theano.foldl`(*fn*, *sequences*, *outputs_info*, *non_sequences=None*, *mode=None*, *name=None*)

Similar behaviour as haskell's foldl.

fn The function that `foldl` applies at each iteration step (see `scan` for more info).

sequences List of sequences over which `foldl` iterates (see `scan` for more info).

outputs_info List of dictionaries describing the outputs of reduce (see `scan` for more info).

non_sequences List of arguments passed to *fn*. `foldl` will not iterate over these arguments (see `scan` for more info).

mode See `scan`.

name See `scan`.

`theano.`**`foldr`**(*fn*, *sequences*, *outputs_info*, *non_sequences=None*, *mode=None*, *name=None*)
Similar behaviour as haskell' foldr.

fn The function that `foldr` applies at each iteration step (see `scan` for more info).

sequences List of sequences over which `foldr` iterates (see `scan` for more info).

outputs_info List of dictionaries describing the outputs of reduce (see `scan` for more info).

non_sequences List of arguments passed to *fn*. `foldr` will not iterate over these arguments (see `scan` for more info).

mode See `scan`.

name See `scan`.

`theano.`**`scan`**(*fn*, *sequences=None*, *outputs_info=None*, *non_sequences=None*, *n_steps=None*, *truncate_gradient=-1*, *go_backwards=False*, *mode=None*, *name=None*, *profile=False*, *allow_gc=None*, *strict=False*)
This function constructs and applies a Scan op to the provided arguments.

fn `fn` is a function that describes the operations involved in one step of `scan`. `fn` should construct variables describing the output of one iteration step. It should expect as input theano variables representing all the slices of the input sequences and previous values of the outputs, as well as all other arguments given to scan as `non_sequences`. The order in which scan passes these variables to `fn` is the following :

- all time slices of the first sequence
- all time slices of the second sequence
- ...
- all time slices of the last sequence
- all past slices of the first output
- all past slices of the second otuput
- ...
- all past slices of the last output
- **all other arguments (the list given as *non_sequences* to** scan)

The order of the sequences is the same as the one in the list *sequences* given to scan. The order of the outputs is the same as the order of `outputs_info`. For any sequence or output the order

of the time slices is the same as the one in which they have been given as taps. For example if one writes the following :

```
scan(fn, sequences = [ dict(input= Sequence1, taps = [-3,2,-1])
                     , Sequence2
                     , dict(input =  Sequence3, taps = 3) ]
       , outputs_info = [ dict(initial =  Output1, taps = [-3,-5])
                        , dict(initial = Output2, taps = None)
                        , Output3 ]
       , non_sequences = [ Argument1, Argument2])
```

`fn` should expect the following arguments in this given order:

1. `Sequence1[t-3]`
2. `Sequence1[t+2]`
3. `Sequence1[t-1]`
4. `Sequence2[t]`
5. `Sequence3[t+3]`
6. `Output1[t-3]`
7. `Output1[t-5]`
8. `Output3[t-1]`
9. `Argument1`
10. `Argument2`

The list of `non_sequences` can also contain shared variables used in the function, though `scan` is able to figure those out on its own so they can be skipped. For the clarity of the code we recommend though to provide them to scan. To some extend `scan` can also figure out other `non sequences` (not shared) even if not passed to scan (but used by *fn*). A simple example of this would be :

```
import theano.tensor as TT
W   = TT.matrix()
W_2 = W**2
def f(x):
    return TT.dot(x,W_2)
```

The function is expected to return two things. One is a list of outputs ordered in the same order as `outputs_info`, with the difference that there should be only one output variable per output initial state (even if no tap value is used). Secondly *fn* should return an update dictionary (that tells how to update any shared variable after each iteration step). The dictionary can optionally be given as a list of tuples. There is no constraint on the order of these two list, `fn` can return either `(outputs_list, update_dictionary)` or `(update_dictionary, outputs_list)` or just one of the two (in case the other is empty).

To use `scan` as a while loop, the user needs to change the function `fn` such that also a stopping condition is returned. To do so, he/she needs to wrap the condition in an `until` class. The condition should be returned as a third element, for example:

```
...
return [y1_t, y2_t], {x:x+1}, theano.scan_module.until(x < 50)
```

Note that a number of steps (considered in here as the maximum number of steps) is still required even though a condition is passed (and it is used to allocate memory if needed). = {}):

sequences `sequences` is the list of Theano variables or dictionaries describing the sequences `scan` has to iterate over. If a sequence is given as wrapped in a dictionary, then a set of optional information can be provided about the sequence. The dictionary should have the following keys:

- `input` (*mandatory*) – Theano variable representing the sequence.
- `taps` – Temporal taps of the sequence required by `fn`. They are provided as a list of integers, where a value `k` impiles that at iteration step `t` scan will pass to `fn` the slice `t+k`. Default value is `[0]`

Any Theano variable in the list `sequences` is automatically wrapped into a dictionary where `taps` is set to `[0]`

outputs_info `outputs_info` is the list of Theano variables or dictionaries describing the initial state of the outputs computed recurrently. When this initial states are given as dictionary optional information can be provided about the output corresponding to these initial states. The dictionary should have the following keys:

- `initial` – Theano variable that represents the initial state of a given output. In case the output is not computed recursively (think of a map) and does not require an initial state this field can be skipped. Given that (only) the previous time step of the output is used by `fn`, the initial state **should have the same shape** as the output and **should not involve a downcast** of the data type of the output. If multiple time taps are used, the initial state should have one extra dimension that should cover all the possible taps. For example if we use `-5`, `-2` and `-1` as past taps, at step 0, `fn` will require (by an abuse of notation) `output[-5]`, `output[-2]` and `output[-1]`. This will be given by the initial state, which in this case should have the shape (5,)+output.shape. If this variable containing the initial state is called `init_y` then `init_y[0]` *corresponds to* `output[-5]`. `init_y[1]` *correponds to* `output[-4]`, `init_y[2]` corresponds to `output[-3]`, `init_y[3]` coresponds to `output[-2]`, `init_y[4]` corresponds to `output[-1]`. While this order might seem strange, it comes natural from splitting an array at a given point. Assume that we have a array `x`, and we choose `k` to be time step `0`. Then our initial state would be `x[:k]`, while the output will be `x[k:]`. Looking at this split, elements in `x[:k]` are ordered exactly like those in `init_y`.
- `taps` – Temporal taps of the output that will be pass to `fn`. They are provided as a list of *negative* integers, where a value `k` implies that at iteration step `t` scan will pass to `fn` the slice `t+k`.

`scan` will follow this logic if partial information is given:

- If an output is not wrapped in a dictionary, `scan` will wrap it in one assuming that you use only the last step of the output (i.e. it makes your tap value list equal to [-1]).
- If you wrap an output in a dictionary and you do not provide any taps but you provide an initial state it will assume that you are using only a tap value of -1.

- If you wrap an output in a dictionary but you do not provide any initial state, it assumes that you are not using any form of taps.
- If you provide a `None` instead of a variable or a empty dictionary `scan` assumes that you will not use any taps for this output (like for example in case of a map)

If `outputs_info` is an empty list or None, `scan` assumes that no tap is used for any of the outputs. If information is provided just for a subset of the outputs an exception is raised (because there is no convention on how scan should map the provided information to the outputs of `fn`)

non_sequences `non_sequences` is the list of arguments that are passed to `fn` at each steps. One can opt to exclude variable used in `fn` from this list as long as they are part of the computational graph, though for clarity we encourage not to do so.

n_steps `n_steps` is the number of steps to iterate given as an int or Theano scalar. If any of the input sequences do not have enough elements, scan will raise an error. If the *value is 0* the outputs will have *0 rows*. If the value is negative, `scan` will run backwards in time. If the `go_backwards` flag is already set and also `n_steps` is negative, `scan` will run forward in time. If n_steps is not provided, `scan` will figure out the amount of steps it should run given its input sequences.

truncate_gradient `truncate_gradient` is the number of steps to use in truncated BPTT. If you compute gradients through a scan op, they are computed using backpropagation through time. By providing a different value then -1, you choose to use truncated BPTT instead of classical BPTT, where you go for only `truncate_gradient` number of steps back in time.

go_backwards `go_backwards` is a flag indicating if `scan` should go backwards through the sequences. If you think of each sequence as indexed by time, making this flag True would mean that `scan` goes back in time, namely that for any sequence it starts from the end and goes towards 0.

name When profiling `scan`, it is crucial to provide a name for any instance of `scan`. The profiler will produce an overall profile of your code as well as profiles for the computation of one step of each instance of `scan`. The `name` of the instance appears in those profiles and can greatly help to disambiguate information.

mode It is recommended to leave this argument to None, especially when profiling `scan` (otherwise the results are not going to be accurate). If you prefer the computations of one step of `scan` to be done differently then the entire function, you can use this parameter to describe how the computations in this loop are done (see `theano.function` for details about possible values and their meaning).

profile Flag or string. If true, or different from the empty string, a profile object will be created and attached to the inner graph of scan. In case `profile` is True, the profile object will have the name of the scan instance, otherwise it will have the passed string. Profile object collect (and print) information only when running the inner graph with the new cvm linker (with default modes, other linkers this argument is useless)

allow_gc Set the value of allow gc for the internal graph of scan. If set to None, this will use the value of config.scan.allow_gc.

strict If true, all the shared variables used in `fn` must be provided as a part of `non_sequences` or `sequences`.

tuple Tuple of the form (outputs, updates); `outputs` is either a Theano variable or a list of Theano variables representing the outputs of `scan` (in the same order as in `outputs_info`). `updates` is a subclass of dictionary specifying the update rules for all shared variables used in scan. This dictionary should be passed to `theano.function` when you compile your function. The change compared to a normal dictionary is that we validate that keys are SharedVariable and addition of those dictionary are validated to be consistent.

`sandbox` – Experimental Code

`sandbox.cuda` – The CUDA GPU backend

`sandbox.cuda` – List of CUDA GPU Op implemented Normally you should not call directly those Ops! Theano should automatically transform cpu ops to their gpu equivalent. So this list is just useful to let people know what is implemented on the gpu.

Basic Op

class `theano.sandbox.cuda.basic_ops.`**`CopyOnNegativeStrides`**(*use_c_code='/usr/bin/g++'*)
Checks if the input has contains negative strides.

If it does, returns a c contiguous copy.

class `theano.sandbox.cuda.basic_ops.`**`GpuAdvancedIncSubtensor1`**(*inplace=False*, *set_instead_of_inc=False*)
Implement AdvancedIncSubtensor1 on the gpu.

class `theano.sandbox.cuda.basic_ops.`**`GpuAdvancedIncSubtensor1_dev20`**(*inplace=False*, *set_instead_of_inc=False*)
Implement AdvancedIncSubtensor1 on the gpu, but use function only avail on compute capability 2.0 and more recent.

`make_node`(*x*, *y*, *ilist*)
It defer from GpuAdvancedIncSubtensor1 in that it make sure the index are of type long.

class `theano.sandbox.cuda.basic_ops.`**`GpuAdvancedSubtensor1`**(*sparse_grad=False*)
Implement AdvancedSubtensor1 on the gpu.

class `theano.sandbox.cuda.basic_ops.`**`GpuAlloc`**(*memset_0=False*)
Implement Alloc on the gpu.

The memset_0 param is an optimization. When True, we call cudaMemset that is faster.

class `theano.sandbox.cuda.basic_ops.`**`GpuAllocEmpty`**(*use_c_code='/usr/bin/g++'*)
Implement Alloc on the gpu, but without initializing memory.

class `theano.sandbox.cuda.basic_ops.`**`GpuCAReduce`**(*reduce_mask*, *scalar_op*, *pre_scalar_op=None*)
GpuCAReduce is a Reduction along some dimensions by a scalar op.

The dimensions along which to reduce is specified by the *reduce_mask* that you pass to the constructor. The *reduce_mask* is a tuple of booleans (actually integers 0 or 1) that specify for each input dimension, whether to reduce it (1) or not (0).

pre_scalar_op If present, must be a scalar op with only 1 input. We will execute it on the input value before reduction.

This Op is a work in progress.

This op was recently upgraded from just GpuSum a general CAReduce. Not many code cases are supported for scalar_op being anything other than scal. Add instances yet.

Important note: if you implement new cases for this op, be sure to benchmark them and make sure that they actually result in a speedup. GPUs are not especially well-suited to reduction operations so it is quite possible that the GPU might be slower for some cases.

When scalar_op is a theano.scalar.basic.Add instance:

- reduce_mask == (1,) sums a vector to a scalar
- reduce_mask == (1,0) computes the sum of each column in a matrix
- reduce_mask == (0,1) computes the sum of each row in a matrix
- reduce_mask == (1,1,1) computes the sum of all elements in a 3-tensor.

..note:: Any reduce_mask of all zeros is a sort of 'copy', and may be removed during graph optimization.

c_code_reduce_01X(*sio*, *node*, *name*, *x*, *z*, *fail*, *N*)

N [int] The number of 1 in the pattern N=1 -> 01, N=2 -> 011 N=3 ->0111 Works for N=1,2,3.

c_code_reduce_ccontig(*sio*, *node*, *name*, *x*, *z*, *fail*)
WRITEME

IG: I believe, based on how this is called in c_code, that it is for the case where we are reducing on all axes and x is C contiguous.

supports_c_code(*inputs*)
Returns True if the current op and reduce pattern has functioning C code.

class theano.sandbox.cuda.basic_ops.**GpuContiguous**(*use_c_code='/usr/bin/g++'*)
Always return a c contiguous output. Copy the input only if it is not already c contiguous.

class theano.sandbox.cuda.basic_ops.**GpuDimShuffle**(*input_broadcastable*, *new_order*)
Implement DimShuffle on the gpu.

class theano.sandbox.cuda.basic_ops.**GpuElemwise**(*scalar_op*, *inplace_pattern=None*, *sync=None*)
Implement a generic elemwise on the gpu.

class theano.sandbox.cuda.basic_ops.**GpuFlatten**(*outdim=1*)
Implement Flatten on the gpu.

class theano.sandbox.cuda.basic_ops.**GpuFromHost**(*use_c_code='/usr/bin/g++'*)
Implement the transfer from cpu to the gpu.

class theano.sandbox.cuda.basic_ops.**GpuIncSubtensor**(*idx_list*, *inplace=False*, *set_instead_of_inc=False*, *destroyhandler_tolerate_aliased=None*)

Implement IncSubtensor on the gpu.

The optimization to make this inplace is in tensor/opt. The same optimization handles IncSubtensor and GpuIncSubtensor. This Op has c_code too; it inherits tensor.IncSubtensor's c_code. The helper methods like do_type_checking, copy_of_x, etc. specialize the c_code for this Op.

copy_into(*view*, *source*)

view [str] C code expression for an array.

source [str] C code expression for an array

str A C code expression to copy source into view, and 0 on success.

copy_of_x(*x*)

x [str] A string giving the name of a C variable pointing to an array.

str C code expression to make a copy of x.

Base class uses *PyArrayObject* *, subclasses may override for different types of arrays.

do_type_checking(*node*)
Should raise NotImplementedError if c_code does not support the types involved in this node.

get_helper_c_code_args()
Return a dictionary of arguments to use with helper_c_code.

make_view_array(*x*, *view_ndim*)

x [str] A string identifying an array to be viewed.

view_ndim [str] A string specifying the number of dimensions to have in the view. This doesn't need to actually set up the view with the right indexing; we'll do that manually later.

class theano.sandbox.cuda.basic_ops.**GpuJoin**(*use_c_code='/usr/bin/g++'*)
Implement Join on the gpu.

class theano.sandbox.cuda.basic_ops.**GpuReshape**(*ndim*, *name=None*)
Implement Reshape on the gpu.

class theano.sandbox.cuda.basic_ops.**GpuShape**(*use_c_code='/usr/bin/g++'*)
Implement Shape on the gpu.

class theano.sandbox.cuda.basic_ops.**GpuSubtensor**(*idx_list*)
Implement subtensor on the gpu.

class theano.sandbox.cuda.basic_ops.**HostFromGpu**(*use_c_code='/usr/bin/g++'*)
Implement the transfer from gpu to the cpu.

theano.sandbox.cuda.basic_ops.**col**(*name=None*, *dtype=None*)
Return a symbolic column variable (ndim=2, broadcastable=[False,True]).

dtype Numeric type (None means to use theano.config.floatX).

name [str] A name to attach to this variable.

theano.sandbox.cuda.basic_ops.**matrix**(*name=None*, *dtype=None*)
Return a symbolic matrix variable.

dtype Numeric type (None means to use theano.config.floatX).

name A name to attach to this variable.

theano.sandbox.cuda.basic_ops.**row**(*name=None*, *dtype=None*)
Return a symbolic row variable (ndim=2, broadcastable=[True,False]).

dtype Numeric type (None means to use theano.config.floatX).

name [str] A name to attach to this variable.

theano.sandbox.cuda.basic_ops.**scalar**(*name=None*, *dtype=None*)
Return a symbolic scalar variable.

dtype Numeric type (None means to use theano.config.floatX).

name [str] A name to attach to this variable.

theano.sandbox.cuda.basic_ops.**tensor3**(*name=None*, *dtype=None*)
Return a symbolic 3-D variable.

dtype Numeric type (None means to use theano.config.floatX).

name [str] A name to attach to this variable.

theano.sandbox.cuda.basic_ops.**tensor4**(*name=None*, *dtype=None*)
Return a symbolic 4-D variable.

dtype Numeric type (None means to use theano.config.floatX).

name [str] A name to attach to this variable.

theano.sandbox.cuda.basic_ops.**vector**(*name=None*, *dtype=None*)
Return a symbolic vector variable.

dtype Numeric type (None means to use theano.config.floatX).

name A name to attach to this variable.

Blas Op

class theano.sandbox.cuda.blas.**BaseGpuCorr3dMM**(*border_mode='valid'*, *subsample=(1, 1, 1)*, *pad=(0, 0, 0)*)
Base class for *GpuCorr3dMM*, *GpuCorr3dMM_gradWeights* and *GpuCorr3dMM_gradInputs*. Cannot be used directly.

c_code_helper(*bottom*, *weights*, *top*, *direction*, *sub*, *height=None*, *width=None*, *depth=None*)
This generates the C code for GpuCorrMM (direction="forward"), GpuCorrMM_gradWeights (direction="backprop weights"), and GpuCorrMM_gradInputs (direction="backprop inputs").

Depending on the direction, one of bottom, weights, top will receive the output, while the other two serve as inputs.

bottom Variable name of the input images in the forward pass, or the gradient of the input images in backprop wrt. inputs.

weights Variable name of the filters in the forward pass, or the gradient of the filters in backprop wrt. weights.

top Variable name of the output images / feature maps in the forward pass, or the gradient of the outputs in the backprop passes.

direction [{'forward', 'backprop weights', 'backprop inputs'}] "forward" to correlate bottom with weights and store results in top, "backprop weights" to do a valid convolution of bottom with top (swapping the first two dimensions) and store results in weights, and "backprop inputs" to do a full convolution of top with weights (swapping the first two dimensions) and store results in bottom.

sub Dictionary of substitutions useable to help generating the C code.

height If self.subsample[0] != 1, a variable giving the height of the filters for direction="backprop weights" or the height of the input images for direction="backprop inputs". If self.pad == 'half', a variable giving the height of the filters for direction="backprop weights". Ignored otherwise.

width If self.subsample[1] != 1, a variable giving the width of the filters for direction="backprop weights" or the width of the input images for direction="backprop inputs". If self.pad == 'half', a variable giving the width of the filters for direction="backprop weights". Ignored otherwise.

depth If self.subsample[2] != 1, a variable giving the depth of the filters for direction="backprop weights" or the depth of the input images for direction="backprop inputs". If self.pad == 'half', a variable giving the depth of the filters for direction="backprop weights". Ignored otherwise.

flops(*inp*, *outp*)

Useful with the hack in profilemode to print the MFlops

class theano.sandbox.cuda.blas.**BaseGpuCorrMM**(*border_mode='valid'*, *subsample=(1, 1)*, *pad=(0, 0)*)

Base class for *GpuCorrMM*, *GpuCorrMM_gradWeights* and *GpuCorrMM_gradInputs*. Cannot be used directly.

border_mode [{'valid', 'full', 'half'}] Additionally, the padding size could be directly specified by an integer or a pair of integers

subsample Perform subsampling of the output (default: (1, 1)).

pad *deprecated*, now you should always use border_mode.

c_code_helper(*bottom*, *weights*, *top*, *direction*, *sub*, *height=None*, *width=None*)

This generates the C code for GpuCorrMM (direction="forward"), GpuCorrMM_gradWeights (direction="backprop weights"), and GpuCorrMM_gradInputs (direction="backprop inputs"). Depending on the direction, one of bottom, weights, top will receive the output, while the other two serve as inputs.

bottom Variable name of the input images in the forward pass, or the gradient of the input images in backprop wrt. inputs

weights Variable name of the filters in the forward pass, or the gradient of the filters in backprop wrt. weights

top Variable name of the output images / feature maps in the forward pass, or the gradient of the outputs in the backprop passes

direction [{'forward', 'backprop weights', 'backprop inputs'}] "forward" to correlate bottom with weights and store results in top, "backprop weights" to do a valid convolution of bottom with top (swapping the first two dimensions) and store results in weights, and "backprop inputs" to do a full convolution of top with weights (swapping the first two dimensions) and store results in bottom.

sub Dictionary of substitutions useable to help generating the C code.

height If self.subsample[0] != 1, a variable giving the height of the filters for direction="backprop weights" or the height of the input images for direction="backprop inputs". If self.border_mode == 'half', a variable giving the height of the filters for direction="backprop weights". Ignored otherwise.

width If self.subsample[1] != 1, a variable giving the width of the filters for direction="backprop weights" or the width of the input images for direction="backprop inputs". If self.border_mode == 'half', a variable giving the width of the filters for direction="backprop weights". Ignored otherwise.

flops(*inp*, *outp*)

Useful with the hack in profilemode to print the MFlops.

class theano.sandbox.cuda.blas.**GpuConv**(*border_mode*, *subsample=(1, 1)*, *logical_img_hw=None*, *logical_kern_hw=None*, *logical_kern_align_top=True*, *version=-1*, *direction_hint=None*, *verbose=0*, *kshp=None*, *imshp=None*, *max_threads_dim0=None*, *nkern=None*, *bsize=None*, *fft_opt=True*)

Implement the batched and stacked 2d convolution on the gpu.

version Each version of c_code implements many kernel for the convolution. By default we try to guess the best one. You can force one version with this parameter. This parameter is used by the tests.

direction_hint [{'forward', 'bprop weights', 'bprop inputs'}] Serves as a hint for graph optimizers replacing GpuConv by other implementations. If the GpuConv is inserted automatically, we take its value from ConvOp.

verbose For value of 1,2 and 3. Print more information during the execution of the convolution. Mostly used for optimization or debugging.

kshp The size of the kernel. If provided, can generate faster code. If the GpuConv op is automatically inserted, We take its value automatically from the Conv op.

imshp The size of the image. Not used for code generation but allows to select an experimental new version in another repo.

max_threads_dim0 The maximum number of threads for the block size dimensions 0 (blockDim.x) used by the GPU function.

nkern The number of kernels. Not used for this op, but can be used by graph optimizers to select a more optimal convolution implementation. If the GpuConv op is inserted automatically, we take its value from the Conv op.

bsize The batch size. Not used for this op, but can be used by graph optimizers to select a more optimal convolution implementation. If the GpuConv op is inserted automatically, we take its value from the Conv op.

fft_opt Deactivate fft_opt optimization at the op level when set to False. Note that by default fft optimization aren't enabled. See *convolution documentation* to enable them.

flops(*inputs*, *outputs*)
Useful with the hack in profilemode to print the MFlops

class theano.sandbox.cuda.blas.**GpuCorr3dMM**(*border_mode='valid'*, *subsample=(1, 1, 1)*, *pad=(0, 0, 0)*)
GPU correlation implementation using Matrix Multiplication.

border_mode Currently supports "valid" only; "full" can be simulated by setting *pad="full"* (at the cost of performance), or by using *GpuCorrMM_gradInputs*.

subsample The subsample operation applied to each output image. Should be a tuple with 3 elements. *(sv, sh, sl)* is equivalent to *GpuCorrMM(...)(...)[:,:,::sv, ::sh, ::sl]*, but faster. Set to *(1, 1, 1)* to disable subsampling.

pad The width of a border of implicit zeros to pad the input image with. Should be a tuple with 3 elements giving the numbers of rows and columns to pad on each side, or "half" to set the padding to *(kernel_rows // 2, kernel_columns // 2, kernel_depth // 2)*, or "full" to set the padding to *(kernel_rows - 1, kernel_columns - 1, kernel_depth - 1)* at runtime. Set to *(0, 0, 0)* to disable padding.

Currently, the Op requires the inputs, filters and outputs to be C-contiguous. Use `gpu_contiguous` on these arguments if needed.

> **Warning:** For 700 series Nvidia GPUs of compute capability 3.5 and CUDA 5.0 to 6.0, there is a bug in CUBLAS' matrix multiplication function that can make GpuCorrMM or its gradients crash for some input and filter shapes. So if you have a Tesla K20, Tesla K40, Quadro K6000, GeForce GT 640 (DDR5), GeForce GTX 780 (or Ti), GeForce GTX TITAN (or Black or Z) and experience a crash, switching to CUDA 6.5 or CUDA 4.2 should fix it. If this is not possible, changing the input or filter shapes (e.g., the batchsize or number of filters) may also work around the CUBLAS bug.

class theano.sandbox.cuda.blas.**GpuCorr3dMM_gradInputs**(*border_mode='valid'*, *subsample=(1, 1, 1)*, *pad=(0, 0, 0)*)
Gradient wrt. inputs for *GpuCorr3dMM*.

You will not want to use this directly, but rely on Theano's automatic differentiation or graph optimization to use it as needed.

class theano.sandbox.cuda.blas.**GpuCorr3dMM_gradWeights**(*border_mode='valid'*, *subsample=(1, 1, 1)*, *pad=(0, 0, 0)*)

Gradient wrt. filters for *GpuCorr3dMM*.

You will not want to use this directly, but rely on Theano's automatic differentiation or graph optimization to use it as needed.

class theano.sandbox.cuda.blas.**GpuCorrMM**(*border_mode='valid'*, *subsample=(1, 1)*, *pad=(0, 0)*)

GPU correlation implementation using Matrix Multiplication.

border_mode The width of a border of implicit zeros to pad the input with. Must be a tuple with 2 elements giving the numbers of rows and columns to pad on each side, or a single integer to pad the same on all sides, or a string shortcut setting the padding at runtime: `'valid'` for `(0, 0)` (valid convolution, no padding), `'full'` for `(kernel_rows - 1, kernel_columns - 1)` (full convolution), `'half'` for `(kernel_rows // 2, kernel_columns // 2)` (same convolution for odd-sized kernels). Note that the two widths are each applied twice, once per side (left and right, top and bottom).

subsample The subsample operation applied to each output image. Should be a tuple with 2 elements. *(sv, sh)* is equivalent to *GpuCorrMM(...)(...)[:,:,::sv, ::sh]*, but faster. Set to *(1, 1)* to disable subsampling.

pad Deprecated alias for *border_mode*.

Currently, the Op requires the inputs, filters and outputs to be C-contiguous. Use `gpu_contiguous` on these arguments if needed.

You can either enable the Theano flag *optimizer_including=conv_gemm* to automatically replace all convolution operations with *GpuCorrMM* or one of its gradients, or you can use it as a replacement for `conv2d`, called as *GpuCorrMM(subsample=...)(image, filters)*. The latter is currently faster, but note that it computes a correlation – if you need to compute a convolution, flip the filters as *filters[:,:,::-1,::-1]*.

..warning:: For 700 series Nvidia GPUs of compute capability 3.5 and CUDA 5.0 to 6.0, there is a bug in CUBLAS' matrix multiplication function that can make GpuCorrMM or its gradients crash for some input and filter shapes. So if you have a Tesla K20, Tesla K40, Quadro K6000, GeForce GT 640 (DDR5), GeForce GTX 780 (or Ti), GeForce GTX TITAN (or Black or Z) and experience a crash, switching to CUDA 6.5 or CUDA 4.2 should fix it. If this is not possible, changing the input or filter shapes (e.g., the batchsize or number of filters) may also work around the CUBLAS bug.

class theano.sandbox.cuda.blas.**GpuCorrMM_gradInputs**(*border_mode='valid'*, *subsample=(1, 1)*, *pad=(0, 0)*)

Gradient wrt. inputs for *GpuCorrMM*.

You will not want to use this directly, but rely on Theano's automatic differentiation or graph optimization to use it as needed.

class theano.sandbox.cuda.blas.**GpuCorrMM_gradWeights**(*border_mode='valid'*, *subsample=(1, 1)*, *pad=(0, 0)*)

Gradient wrt. filters for *GpuCorrMM*.

You will not want to use this directly, but rely on Theano's automatic differentiation or graph optimization to use it as needed.

class theano.sandbox.cuda.blas.**GpuDot22**(*use_c_code='/usr/bin/g++'*)
Implement dot(2d, 2d) on the gpu.

class theano.sandbox.cuda.blas.**GpuDot22Scalar**(*use_c_code='/usr/bin/g++'*)
Implement dot(2d, 2d) * scalar on the gpu.

Not used anymore. Keep to allow unpickle of old graph.

class theano.sandbox.cuda.blas.**GpuDownsampleFactorMax**(*ds*, *ignore_border=False*)
Implement downsample with max on the gpu.

class theano.sandbox.cuda.blas.**GpuDownsampleFactorMaxGrad**(*ds*, *ignore_border*)
Implement the grad of downsample with max on the gpu.

class theano.sandbox.cuda.blas.**GpuDownsampleFactorMaxGradGrad**(*ds*, *ignore_border*)
Implement the grad of downsample with max on the gpu.

class theano.sandbox.cuda.blas.**GpuGemm**(*inplace*)
implement the gemm on the gpu.

class theano.sandbox.cuda.blas.**GpuGemv**(*inplace*)
implement gemv on the gpu.

class theano.sandbox.cuda.blas.**GpuGer**(*inplace*)
implement ger on the gpu.

Nnet Op

class theano.sandbox.cuda.nnet.**GpuCrossentropySoftmax1HotWithBiasDx**(***kwargs*)
Implement CrossentropySoftmax1HotWithBiasDx on the gpu.

class theano.sandbox.cuda.nnet.**GpuCrossentropySoftmaxArgmax1HotWithBias**(*use_c_code='/usr/*
Implement CrossentropySoftmaxArgmax1HotWithBias on the gpu.

class theano.sandbox.cuda.nnet.**GpuSoftmax**(*use_c_code='/usr/bin/g++'*)
Implement Softmax on the gpu.

class theano.sandbox.cuda.nnet.**GpuSoftmaxWithBias**(*use_c_code='/usr/bin/g++'*)
Implement SoftmaxWithBias on the gpu.

Curand Op Random generator based on the CURAND libraries. It is not inserted automatically. Define CURAND_RandomStreams - backed by CURAND.

class theano.sandbox.cuda.rng_curand.**CURAND_Base**(*output_type*, *seed*, *destructive*)
Base class for a random number generator implemented in CURAND.

The random number generator itself is an opaque reference managed by CURAND. This Op uses a generic-typed shared variable to point to a CObject that encapsulates this opaque reference.

Each random variable is created with a generator of False. The actual random number generator is allocated from the seed, on the first call to allocate random numbers (see c_code).

output_type A theano type (e.g. tensor.fvector).

seed: int destructive

True or False (on the generator)

One caveat is that the random number state is simply not serializable. Consequently, attempts to serialize functions compiled with these random numbers will fail.

as_destructive()
Return an destructive version of self.

classmethod new_auto_update(*generator*, *ndim*, *dtype*, *size*, *seed*)
Return a symbolic sample from generator.

cls dictates the random variable (e.g. uniform, normal).

class theano.sandbox.cuda.rng_curand.**CURAND_Normal**(*output_type*, *seed*, *destructive*)
Op to draw normal numbers using CURAND.

class theano.sandbox.cuda.rng_curand.**CURAND_RandomStreams**(*seed*)
RandomStreams instance that creates CURAND-based random variables.

One caveat is that generators are not serializable.

seed : int

next_seed()
Return a unique seed for initializing a random variable.

normal(*size=None*, *avg=0.0*, *std=1.0*, *ndim=None*, *dtype='float64'*)
Return symbolic tensor of normally-distributed numbers.

size Can be a list of integer or Theano variable (ex: the shape of other Theano Variable)

uniform(*size*, *low=0.0*, *high=1.0*, *ndim=None*, *dtype='float64'*)
Return symbolic tensor of uniform numbers.

updates()
List of all (old, new) generator update pairs created by this instance.

class theano.sandbox.cuda.rng_curand.**CURAND_Uniform**(*output_type*, *seed*, *destructive*)
Op to draw uniform numbers using CURAND.

sandbox.cuda.var – The Variables for Cuda-allocated arrays

API

class theano.sandbox.cuda.var.**CudaNdarraySharedVariable**(*name*, *type*, *value*, *strict*, *allow_downcast=None*, *container=None*)

Shared Variable interface to CUDA-allocated arrays.

get_value(*borrow=False*, *return_internal_type=False*)

Return the value of this SharedVariable's internal array.

borrow Permit the return of internal storage, when used in conjunction with `return_internal_type=True`.

return_internal_type True to return the internal `cuda_ndarray` instance rather than a `numpy.ndarray` (Default False).

By default `get_value()` copies from the GPU to a `numpy.ndarray` and returns that host-allocated array.

`get_value(False,True)` will return a GPU-allocated copy of the original GPU array.

`get_value(True,True)` will return the original GPU-allocated array without any copying.

set_value(*value*, *borrow=False*)

Assign *value* to the GPU-allocated array.

borrow [bool] `True` permits reusing *value* itself, `False` requires that this function copies *value* into internal storage.

Prior to Theano 0.3.1, set_value did not work in-place on the GPU. This meant that sometimes, GPU memory for the new value would be allocated before the old memory was released. If you're running near the limits of GPU memory, this could cause you to run out of GPU memory.

Beginning with Theano 0.3.1, set_value will work in-place on the GPU, if the following conditions are met:

•The destination on the GPU must be c_contiguous.

•The source is on the CPU.

•The old value must have the same dtype as the new value (which is

a given for now, since only float32 is supported). * The old and new value must have the same shape. * The old value is being completely replaced by the new value (not partially modified, e.g. by replacing some subtensor of it). * You change the value of the shared variable via set_value, not via the .value accessors. You should not use the .value accessors anyway, since they will soon be deprecated and removed.

It is also worth mentioning that, for efficient transfer to the GPU, Theano will make the new data `c_contiguous`. This can require an extra copy of the data on the host.

The inplace on gpu memory work when borrow is either True or False.

sandbox.cuda.type – The Type object for Cuda-allocated arrays

API

sandbox.cuda.dnn – cuDNN cuDNN is an NVIDIA library with functionality used by deep neural network. It provides optimized versions of some operations like the convolution. cuDNN is not currently installed with CUDA 6.5. You must download and install it yourself.

To install it, decompress the downloaded file and make the `*.h` and `*.so*` files available to the compilation environment. There are at least three possible ways of doing so:

- The easiest is to include them in your CUDA installation. Copy the `*.h` files to `CUDA_ROOT/include` and the `*.so*` files to `CUDA_ROOT/lib64` (by default, `CUDA_ROOT` is `/usr/local/cuda` on Linux).
- Alternatively, on Linux, you can set the environment variables `LD_LIBRARY_PATH`, `LIBRARY_PATH` and `CPATH` to the directory extracted from the download. If needed, separate multiple directories with `:` as in the `PATH` environment variable.
- And as a third way, also on Linux, you can copy the `*.h` files to `/usr/include` and the `*.so*` files to `/lib64`.

By default, Theano will detect if it can use cuDNN. If so, it will use it. If not, Theano optimizations will not introduce cuDNN ops. So Theano will still work if the user did not introduce them manually.

To get an error if Theano can not use cuDNN, use this Theano flag: `optimizer_including=cudnn`.

Note: CuDNN v3 has now been released. CuDNN v2 remains supported but CuDNN v3 is faster and offers many more options. We recommend that everybody update to v3.

Note: Starting in CuDNN v3, multiple convolution implementations are offered and it is possible to use heuristics to automatically choose a convolution implementation well suited to the parameters of the convolution.

The Theano flag `dnn.conv.algo_fwd` allows to specify the CuDNN convolution implementation that Theano should use for forward convolutions. Possible values include :

- `small` (default) : use a convolution implementation with small memory usage
- `none` : use a slower implementation with minimal memory usage
- `large` : use a sometimes faster implementation with large memory usage
- `fft` : use the Fast Fourrier Transform implementation of convolution (very high memory usage)
- `guess_once` : the first time a convolution is executed, the implementation to use is chosen according to CuDNN's heuristics and reused for every subsequent execution of the convolution.
- `guess_on_shape_change` : like `guess_once` but a new convolution implementation selected every time the shapes of the inputs and kernels don't match the shapes from the last execution.
- `time_once` : the first time a convolution is executed, every convolution implementation offered by CuDNN is executed and timed. The fastest is reused for every subsequent execution of the convolution.

- `time_on_shape_change` : like `time_once` but a new convolution implementation selected every time the shapes of the inputs and kernels don't match the shapes from the last execution.

The Theano flag `dnn.conv.algo_bwd` allows to specify the CuDNN convolution implementation that Theano should use for gradient convolutions. Possible values include :

- `none` (default) : use the default non-deterministic convolution implementation
- `deterministic` : use a slower but deterministic implementation
- `fft` : use the Fast Fourrier Transform implementation of convolution (very high memory usage)
- `guess_once` : the first time a convolution is executed, the implementation to use is chosen according to CuDNN's heuristics and reused for every subsequent execution of the convolution.
- `guess_on_shape_change` : like `guess_once` but a new convolution implementation selected every time the shapes of the inputs and kernels don't match the shapes from the last execution.
- `time_once` : the first time a convolution is executed, every convolution implementation offered by CuDNN is executed and timed. The fastest is reused for every subsequent execution of the convolution.
- `time_on_shape_change` : like `time_once` but a new convolution implementation selected every time the shapes of the inputs and kernels don't match the shapes from the last execution.

`guess_*` and `time_*` flag values take into account the amount of available memory when selecting an implementation. This means that slower implementations might be selected if not enough memory is available for the faster implementations.

Note: Normally you should not call GPU Ops directly, but the CPU interface currently does not allow all options supported by cuDNN ops. So it is possible that you will need to call them manually.

Note: The documentation of CUDNN tells that, for the 2 following operations, the reproducibility is not guaranteed with the default implementation: *cudnnConvolutionBackwardFilter* and *cudnnConvolutionBackwardData*. Those correspond to the gradient wrt the weights and the gradient wrt the input of the convolution. They are also used sometimes in the forward pass, when they give a speed up.

The Theano flag `dnn.conv.algo_bwd` can be use to force the use of a slower but deterministic convolution implementation.

Note: There is a problem we do not understand yet when cudnn paths are used with symbolic links. So avoid using that.

Note: cudnn.so* must be readable and executable by everybody. cudnn.h must be readable by everybody.

Functions

`theano.sandbox.cuda.dnn.`**`dnn_conv`**(*img*, *kerns*, *border_mode='valid'*, *subsample=(1, 1)*, *conv_mode='conv'*, *direction_hint=None*, *workmem=None*, *algo=None*)

GPU convolution using cuDNN from NVIDIA.

The memory layout to use is 'bc01', that is 'batch', 'channel', 'first dim', 'second dim' in that order.

img Images to do the convolution over.

kerns Convolution filters.

border_mode One of 'valid', 'full'; additionally, the padding size could be directly specified by an integer or a pair of integers.

subsample Perform subsampling of the output (default: (1, 1)).

conv_mode Perform convolution (kernels flipped) or cross-correlation. One of 'conv', 'cross' (default: 'conv').

direction_hint Used by graph optimizers to change algorithm choice. By default, GpuDnnConv will be used to carry out the convolution. If border_mode is 'valid', subsample is (1,1) and direction_hint is 'bprop weights', it will use GpuDnnConvGradW. If border_mode is 'full', subsample is (1,1) and direction_hint is 'bprop inputs', it will use GpuDnnConvGradI. This parameter is used internally by graph optimizers and may be removed at any time without a deprecation period. You have been warned.

workmem *deprecated*, use parameter algo instead.

algo [{'none', 'small', 'large', 'fft', 'guess_once', 'guess_on_shape_change', 'time_once', 'time_on_shape_change'}] Convolution implementation to use. Some of its values may require certain versions of CuDNN to be installed. Default is the value of :attr:'config.dnn.conv.algo_fwd.

`theano.sandbox.cuda.dnn.`**`dnn_pool`**(*img*, *ws*, *stride=(1, 1)*, *mode='max'*, *pad=(0, 0)*)

GPU pooling using cuDNN from NVIDIA.

The memory layout to use is 'bc01', that is 'batch', 'channel', 'first dim', 'second dim' in that order.

img Images to do the pooling over.

ws Subsampling window size.

stride Subsampling stride (default: (1, 1)).

mode : {'max', 'average_inc_pad', 'average_exc_pad} pad

(padX, padY) padding information. padX is the size of the left and right borders, padY is the size of the top and bottom borders.

Parameters **nd** – dimensions of pooling, can be 2 or 3 for 2d or 3d pooling If set to 3 all other params (except mode) must have an extra dimension to match. 3 is only available for cudnn v3

Warning: The cuDNN library only works with GPU that have a compute capability of 3.0 or higer. This means that older GPU will not work with this Op.

This Op implements the ignore_border=True of max_pool_2d.

Convolution Ops

class theano.sandbox.cuda.dnn.**GpuDnnConvDesc**(*border_mode*, *subsample=(1, 1)*, *conv_mode='conv'*)

This Op builds a convolution descriptor for use in the other convolution operations.

See the doc of `dnn_conv()` for a description of the parameters.

class theano.sandbox.cuda.dnn.**GpuDnnConv**(*workmem=None*, *inplace=False*, *algo=None*)

The forward convolution.

image kernel descr

The convolution descriptor.

workmem *deprecated*, use parameter algo instead.

algo [{'small', 'none', 'large', 'fft', 'guess_once', 'guess_on_shape_change', 'time_once', 'time_on_shape_change'}] Default is the value of `config.dnn.conv.algo_fwd`.

static GpuDnnConv.**get_out_shape**(*ishape*, *kshape*, *border_mode*, *subsample*)

This function computes the output shape for a convolution with the specified parameters. *ishape* and *kshape* can be symbolic or scalar.

class theano.sandbox.cuda.dnn.**GpuDnnConv3d**(*workmem=None*, *inplace=False*, *algo=None*)

The forward convolution.

Parameters

- **image** –
- **kernel** –
- **descr** – the convolution descriptor

static GpuDnnConv3d.**get_out_shape**(*ishape*, *kshape*, *border_mode*, *subsample*)

This function computes the output shape for a convolution with the specified parameters. *ishape* and *kshape* can be symbolic or scalar.

class theano.sandbox.cuda.dnn.**GpuDnnConvGradW**(*inplace=False*, *workmem=None*, *algo=None*)

The convolution gradient with respect to the weights.

image kernel descr

The convolution descriptor.

class theano.sandbox.cuda.dnn.**GpuDnnConv3dGradW**(*inplace=False*, *workmem=None*, *algo=None*)

The convolution gradient with respect to the weights.

Parameters

- **image** –
- **kernel** –
- **descr** – the convolution descriptor

class theano.sandbox.cuda.dnn.**GpuDnnConvGradI**(*inplace=False*, *workmem=None*, *algo=None*)

The convolution gradient with respect to the inputs.

image kernel descr

The convolution descriptor.

workmem *deprecated*, use parameter algo instead.

algo [{'none', 'deterministic', 'fft', 'guess_once', 'guess_on_shape_change', 'time_once', 'time_on_shape_change'}] Default is the value of theano.config.dnn.conv.algo_bwd.

class theano.sandbox.cuda.dnn.**GpuDnnConv3dGradI**(*inplace=False*, *workmem=None*, *algo=None*)

The convolution gradient with respect to the inputs.

Parameters

- **image** –
- **kernel** –
- **descr** – the convolution descriptor

Pooling Ops

class theano.sandbox.cuda.dnn.**GpuDnnPoolDesc**(*ws=(1, 1)*, *stride=(1, 1)*, *mode='max'*, *pad=(0, 0)*)

This Op builds a pooling descriptor for use in the other pooling operations.

ws Windows size.

stride (dx, dy).

mode [{'max', 'average_inc_pad', 'average_exc_pad'}] The old deprecated name 'average' correspond to 'average_inc_pad'.

pad (padX, padY) padding information. padX is the size of the left and right borders, padY is the size of the top and bottom borders.

class theano.sandbox.cuda.dnn.**GpuDnnPool**

Pooling.

img The image 4d or 5d tensor.

desc The pooling descriptor.

class theano.sandbox.cuda.dnn.**GpuDnnPoolGrad**

The pooling gradient.

inp The input of the pooling.

out The output of the pooling in the forward.

inp_grad Same size as out, but is the corresponding gradient information.

desc The pooling descriptor.

Softmax Ops

class theano.sandbox.cuda.dnn.**GpuDnnSoftmax**(*tensor_format*, *algo*, *mode*)
Op for the cuDNN Softmax.

tensor_format Whether the data format is 'bc01' or 'b01c'.

algo 'fast' or 'accurate' indicating whether computations should be optimized for speed or accuracy respectively.

mode 'instance' or 'channel' indicating whether the softmax should be computed per image across 'c01' or per spatial location '01' per image across 'c'.

class theano.sandbox.cuda.dnn.**GpuDnnSoftmaxGrad**(*tensor_format*, *algo*, *mode*)
Op for the cuDNN SoftmaxGrad.

tensor_format Whether the data format is 'bc01' or 'b01c'.

algo 'fast' or 'accurate' indicating whether computations should be optimized for speed or accuracy respectively.

mode 'instance' or 'channel' indicating whether the softmax should be computed per image across 'c01' or per spatial location '01' per image across 'c'.

sandbox.linalg – Linear Algebra Ops

API

class theano.sandbox.linalg.ops.**Hint**(***kwargs*)
Provide arbitrary information to the optimizer.

These ops are removed from the graph during canonicalization in order to not interfere with other optimizations. The idea is that prior to canonicalization, one or more Features of the fgraph should register the information contained in any Hint node, and transfer that information out of the graph.

class theano.sandbox.linalg.ops.**HintsFeature**
FunctionGraph Feature to track matrix properties.

This is a similar feature to variable 'tags'. In fact, tags are one way to provide hints.

This class exists because tags were not documented well, and the semantics of how tag information should be moved around during optimizations was never clearly spelled out.

Hints are assumptions about mathematical properties of variables. If one variable is substituted for another by an optimization, then it means that the assumptions should be transferred to the new variable.

Hints are attached to 'positions in a graph' rather than to variables in particular, although Hints are originally attached to a particular posititon in a graph *via* a variable in that original graph.

Examples of hints are: - shape information - matrix properties (e.g. symmetry, psd, banded, diagonal)

Hint information is propagated through the graph similarly to graph optimizations, except that adding a hint does not change the graph. Adding a hint is not something that debugmode will check.

#TODO: should a Hint be an object that can actually evaluate its # truthfulness? # Should the PSD property be an object that can check the # PSD-ness of a variable?

class theano.sandbox.linalg.ops.**HintsOptimizer**
Optimizer that serves to add HintsFeature as an fgraph feature.

theano.sandbox.linalg.ops.**psd**(*v*)
Apply a hint that the variable *v* is positive semi-definite, i.e. it is a symmetric matrix and $x^T Ax \geq 0$ for any vector x.

theano.sandbox.linalg.ops.**spectral_radius_bound**(*X*, *log2_exponent*)
Returns upper bound on the largest eigenvalue of square symmetrix matrix X.

log2_exponent must be a positive-valued integer. The larger it is, the slower and tighter the bound. Values up to 5 should usually suffice. The algorithm works by multiplying X by itself this many times.

From V.Pan, 1990. "Estimating the Extremal Eigenvalues of a Symmetric Matrix", Computers Math Applic. Vol 20 n. 2 pp 17-22. Rq: an efficient algorithm, not used here, is defined in this paper.

sandbox.neighbours – Neighbours Ops

API Neighbours was moved into theano.tensor.nnet.neighbours. This file was created for compatibility.

theano.sandbox.neighbours.**images2neibs**(*ten4*, *neib_shape*, *neib_step=None*, *mode='valid'*)
Function images2neibs allows to apply a sliding window operation to a tensor containing images or other two-dimensional objects. The sliding window operation loops over points in input data and stores a rectangular neighbourhood of each point. It is possible to assign a step of selecting patches (parameter *neib_step*).

Parameters

- **ten4** (*A 4d tensor-like.*) – A 4-dimensional tensor which represents a list of lists of images.a list of lists of images. It should have shape (list 1 dim, list 2 dim, row, col). The first two dimensions can be useful to store different channels and batches.
- **neib_shape** (*A 1d tensor-like of 2 values.*) – A tuple containing two values: height and width of the neighbourhood. It should have shape (r,c) where r is the height of the neighborhood in rows and c is the width of the neighborhood in columns
- **neib_step** (*A 1d tensor-like of 2 values.*) – (dr,dc) where dr is the number of rows to skip between patch and dc is the number of columns. The parameter should be a tuple of two elements: number of rows and number of columns to skip each iteration. Basically, when the step is 1, the neighbourhood of every first element is taken and every possible rectangular subset is returned. By default it is equal to *neib_shape* in other words, the patches are disjoint. When the step is greater than *neib_shape*, some elements are omitted. When None, this is the same as neib_shape(patch are disjoint)

Note: Currently the step size should be chosen in the way that the corresponding dimension i (width or height) is equal to $n * step\_size_i + neib\_shape_i$ for some n

---

- **mode** (*str*) – Possible values:

  valid Requires an input that is a multiple of the pooling factor (in each direction)

  ignore_borders Same as valid, but will ignore the borders if the shape(s) of the input is not a multiple of the pooling factor(s)

  wrap_centered ?? TODO comment

Returns

Reshapes the input as a 2D tensor where each row is an pooling example. Pseudo-code of the output:

```
idx = 0
for i in xrange(list 1 dim):
    for j in xrange(list 2 dim):
        for k in <image column coordinates>:
            for l in <image row coordinates>:
                output[idx,:]
                     = flattened version of ten4[i,j,l:l+r,k:k+c]
                idx += 1
```

Note: The operation isn't necessarily implemented internally with these for loops, they're just the easiest way to describe the output pattern.

Example:

```
# Defining variables
images = T.tensor4('images')
neibs = images2neibs(images, neib_shape=(5, 5))

# Constructing theano function
window_function = theano.function([images], neibs)

# Input tensor (one image 10x10)
im_val = np.arange(100.).reshape((1, 1, 10, 10))

# Function application
neibs_val = window_function(im_val)
```

Note: The underlying code will construct a 2D tensor of disjoint patches 5x5. The output has shape 4x25.

`theano.sandbox.neighbours.`**`neibs2images`**(*neibs*, *neib_shape*, *original_shape*, *mode='valid'*)

Function `neibs2images` performs the inverse operation of `images2neibs`. It inputs the output of `images2neibs` and reconstructs its input.

Parameters

- **neibs** – matrix like the one obtained by `images2neibs`

- **neib_shape** – *neib_shape* that was used in `images2neibs`
- **original_shape** – original shape of the 4d tensor given to `images2neibs`

Returns Reconstructs the input of `images2neibs`, a 4d tensor of shape *original_shape*.

Note: Currently, the function doesn't support tensors created with *neib_step* different from default value. This means that it may be impossible to compute the gradient of a variable gained by `images2neibs` w.r.t. its inputs in this case, because it uses `images2neibs` for gradient computation.

Example, which uses a tensor gained in example for `images2neibs`:

```
im_new = neibs2images(neibs, (5, 5), im_val.shape)
# Theano function definition
inv_window = theano.function([neibs], im_new)
# Function application
im_new_val = inv_window(neibs_val)
```

Note: The code will output the initial image array.

`sandbox.rng_mrg` – MRG random number generator

API Implementation of MRG31k3p random number generator for Theano.

Generator code in SSJ package (L'Ecuyer & Simard). http://www.iro.umontreal.ca/~simardr/ssj/indexe.html

class `theano.sandbox.rng_mrg.`**`DotModulo`**(*use_c_code='/usr/bin/g++'*)

Efficient and numerically stable implementation of a dot product followed by a modulo operation. This performs the same function as matVecModM.

We do this 2 times on 2 triple inputs and concatenating the output.

class `theano.sandbox.rng_mrg.`**`MRG_RandomStreams`**(*seed=12345*, *use_cuda=None*)

Module component with similar interface to numpy.random (numpy.random.RandomState).

seed [int or list of 6 int] A default seed to initialize the random state. If a single int is given, it will be replicated 6 times. The first 3 values of the seed must all be less than M1 = 2147483647, and not all 0; and the last 3 values must all be less than M2 = 2147462579, and not all 0.

`get_substream_rstates`(*n_streams*, *dtype*, *inc_rstate=True*)

Initialize a matrix in which each row is a MRG stream state, and they are spaced by 2**72 samples.

`inc_rstate`()

Update self.rstate to be skipped 2^134 steps forward to the next stream start.

`multinomial`(*size=None*, *n=1*, *pvals=None*, *ndim=None*, *dtype='int64'*, *nstreams=None*)

Sample *n* (currently *n* needs to be 1) times from a multinomial distribution defined by probabilities pvals.

Example : pvals = [[.98, .01, .01], [.01, .98, .01]] will probably result in [[1,0,0],[0,1,0]].

-*size* and *ndim* are only there keep the same signature as other uniform, binomial, normal, etc. TODO : adapt multinomial to take that into account

-Does not do any value checking on pvals, i.e. there is no check that the elements are non-negative, less than 1, or sum to 1. passing pvals = [[-2., 2.]] will result in sampling [[0, 0]]

normal(*size*, *avg=0.0*, *std=1.0*, *ndim=None*, *dtype=None*, *nstreams=None*)

size Can be a list of integers or Theano variables (ex: the shape of another Theano Variable).

dtype The output data type. If dtype is not specified, it will be inferred from the dtype of low and high, but will be at least as precise as floatX.

nstreams Number of streams.

seed(*seed=None*)

Re-initialize each random stream.

seed [None or integer in range 0 to 2**30] Each random stream will be assigned a unique state that depends deterministically on this value.

None

uniform(*size*, *low=0.0*, *high=1.0*, *ndim=None*, *dtype=None*, *nstreams=None*)

Sample a tensor of given size whose element from a uniform distribution between low and high.

If the size argument is ambiguous on the number of dimensions, ndim may be a plain integer to supplement the missing information.

low Lower bound of the interval on which values are sampled. If the `dtype` arg is provided, `low` will be cast into dtype. This bound is excluded.

high Higher bound of the interval on which values are sampled. If the `dtype` arg is provided, `high` will be cast into dtype. This bound is excluded.

size Can be a list of integer or Theano variable (ex: the shape of other Theano Variable).

dtype The output data type. If dtype is not specified, it will be inferred from the dtype of low and high, but will be at least as precise as floatX.

theano.sandbox.rng_mrg.**guess_n_streams**(*size*, *warn=False*)

Return a guess at a good number of streams.

warn [bool, optional] If True, warn when a guess cannot be made (in which case we return 60 * 256).

theano.sandbox.rng_mrg.**multMatVect**(*v*, *A*, *m1*, *B*, *m2*)

Multiply the first half of v by A with a modulo of m1 and the second half by B with a modulo of m2.

The parameters of dot_modulo are passed implicitly because passing them explicitly takes more time than running the function's C-code.

`typed_list` – Typed List

Note: This has been added in release 0.7.

Note: This works, but is not well integrated with the rest of Theano. If speed is important, it is probably better to pad to a dense tensor.

This is a type that represents a list in Theano. All elements must have the same Theano type. Here is an example:

```
import theano.typed_list

tl = theano.typed_list.TypedListType(theano.tensor.fvector)()
v = theano.tensor.fvector()
o = theano.typed_list.append(tl, v)
f = theano.function([tl, v], o)
print f([[1, 2, 3], [4, 5]], [2])
#[array([ 1.,  2.,  3.], dtype=float32), array([ 4.,  5.], dtype=float32), array([ 2.], dty
```

A second example with Scan. Scan doesn't yet have direct support of TypedList, so you can only use it as non_sequences (not in sequences or as outputs):

```
import theano.typed_list

a = theano.typed_list.TypedListType(theano.tensor.fvector)()
l = theano.typed_list.length(a)
s, _ = theano.scan(fn=lambda i, tl: tl[i].sum(),
                   non_sequences=[a],
                   sequences=[theano.tensor.arange(l, dtype='int64')])

f = theano.function([a], s)
f([[1, 2, 3], [4, 5]])
#array([ 6.,  9.], dtype=float32)
```

class theano.typed_list.basic.**TypedListVariable**(*type*, *owner=None*, *index=None*, *name=None*)

Subclass to add the typed list operators to the basic *Variable* class.

theano.typed_list.basic.**append = <theano.typed_list.basic.Append object at 0x7e3f0d0>**

Append an element at the end of another list.

x The base typed list.

y The element to append to *x*.

theano.typed_list.basic.**count = <theano.typed_list.basic.Count object at 0x7e3f190>**

Count the number of times an element is in the typed list.

x The typed list to look into.

elem The element we want to count in list. The elements are compared with equals.

Python implementation of count doesn't work when we want to count an ndarray from a list. This implementation works in that case.

theano.typed_list.basic.**extend = <theano.typed_list.basic.Extend object at 0x7e3f150>**

Append all elements of a list at the end of another list.

x The typed list to extend.

toAppend The typed list that will be added at the end of *x*.

theano.typed_list.basic.**getitem = <theano.typed_list.basic.GetItem object at 0x7e3fe10>**
Get specified slice of a typed list.

x Typed list.

index The index of the value to return from *x*.

theano.typed_list.basic.**insert = <theano.typed_list.basic.Insert object at 0x7e3ff90>**
Insert an element at an index in a typed list.

x The typed list to modify.

index The index where to put the new element in *x*.

toInsert The new element to insert.

theano.typed_list.basic.**length = <theano.typed_list.basic.Length object at 0x7e3fd10>**
Returns the size of a list.

x Typed list.

theano.typed_list.basic.**make_list = <theano.typed_list.basic.MakeList object at 0x7e3f5d0>**
Build a Python list from those Theano variable.

a : tuple/list of Theano variable

All Theano variables must have the same type.

theano.typed_list.basic.**remove = <theano.typed_list.basic.Remove object at 0x7e3f290>**
Remove an element from a typed list.

x The typed list to be changed.

toRemove An element to be removed from the typed list. We only remove the first instance.

Python implementation of remove doesn't work when we want to remove an ndarray from a list. This implementation works in that case.

theano.typed_list.basic.**reverse = <theano.typed_list.basic.Reverse object at 0x7e3fad0>**
Reverse the order of a typed list.

x The typed list to be reversed.

There are also some top-level imports that you might find more convenient:

theano.**function**(...)
Alias for `function.function()`

theano.**function_dump**(...)
Alias for `theano.compile.function.function_dump()`

theano.**shared**(...)
Alias for `theano.compile.sharedvalue.shared()`

class theano.**Param**
Alias for `function.Param`

theano.**dot**(*x*, *y*)
Works like tensor.dot() for both sparse and dense matrix products

theano.**clone**(*output*, *replace=None*, *strict=True*, *share_inputs=True*, *copy_inputs=<object object at 0x2905d70>*)
Function that allows replacing subgraphs of a computational graph.

It returns a copy of the initial subgraph with the corresponding substitutions.

output [Theano Variables (or Theano expressions)] Theano expression that represents the computational graph.

replace [dict] Dictionary describing which subgraphs should be replaced by what.

share_inputs [bool] If True, use the same inputs (and shared variables) as the original graph. If False, clone them. Note that cloned shared variables still use the same underlying storage, so they will always have the same value.

copy_inputs Deprecated, use share_inputs.

theano.**sparse_grad**(*var*)
This function return a new variable whose gradient will be stored in a sparse format instead of dense.

Currently only variable created by AdvancedSubtensor1 is supported. i.e. a_tensor_var[an_int_vector].

New in version 0.6rc4.

6.2.6 Optimizations

Theano applies many kinds of graph optimizations, with different objectives:

- simplifying and standardizing the form of the expression graph (e.g. *merge*, *add canonicalization*),
- reducing the maximum memory footprint (e.g. *inplace_elemwise*),
- increasing execution speed (e.g. *constant folding*).

The optimizations are listed in roughly chronological order. The table below gives a quick summary of the optimizations included in the default modes. The descriptions are brief and point to further reading.

If you would like to add an additional optimization, refer to *Graph optimization* in the guide to extending Theano.

Note: This list is partial.

The print_summary method allows several OpDBs and optimizers to list the executed optimizations. This makes it possible to have an up-to-date list.

python -c 'import theano; theano.compile.FAST_RUN.optimizer.print_summary()'

python -c 'import theano; theano.compile.FAST_COMPILE.optimizer.print_summary()'

Optimization	FAST_RUN	FAST_COMPILE	Stabilization
merge	x	x	
constant folding	x	x	
shape promotion	x		
fill cut	x		
inc_subtensor srlz.	x		
reshape_chain	x		
const. elimination	x		
add canonical.	x		
mul canonical.	x		
dot22	x		
sparse_dot	x		
sum_scalar_mul	x		
neg_neg	x		
neg_div_neg	x		
add specialize	x		
mul specialize	x		
pow specialize	x		
inplace_setsubtensor	x		
gemm	x		
inplace_elemwise	x		
inplace_random	x		
elemwise fusion	x		
GPU transfer	x		
local_log_softmax	x		x
local_remove_all_assert			

merge A simple optimization in which redundant *Apply* nodes are combined. For example, in `function([x,y], [(x+y)*2, (x+y)*3])` the merge optimization will ensure that `x` and `y` are only added once.

This optimization is very useful because it frees users to write highly redundant mathematical code. Theano will make sure to compute just what is necessary.

See `MergeOptimizer`.

constant folding When all the inputs to an expression are constant, then the expression can be pre-computed at compile-time.

See `opt.constant_folding()`

shape promotion Theano often knows how to infer the shape of an output from the shape of its inputs. Without this optimization, it would otherwise have to compute things (e.g. `log(x)`) just to find out the shape of it!

See `opt.local_shape_lift_*()`

fill cut *Fill(a,b)* means to make a tensor of the shape of *a* full of the value *b*. Often when fills are used with elementwise operations (e.g. f) they are un-necessary: * `f(fill(a,b), c) -> f(b, c)` * `f(fill(a, b), fill(c, d), e) -> fill(a, fill(c, f(b, d, e)))`

See `opt.local_fill_cut()`, `opt.local_fill_sink()`

inc_subtensor serialization Incrementing a small subregion of a large tensor can be done quickly using an inplace operation, but if two increments are being done on the same large tensor, then only one of them can be done inplace. This optimization reorders such graphs so that all increments can be done inplace.

```
inc_subensor(a,b,idx) + inc_subtensor(a,c,idx) ->
inc_subtensor(inc_subtensor(a,b,idx),c,idx)
```

See `local_IncSubtensor_serialize()`

reshape_chain This optimizes graphs like `reshape(reshape(x, shape1), shape2) -> reshape(x, shape2)`

See `local_reshape_chain()`

constant elimination Many constants indicate special cases, such as `pow(x,1) -> x`. Theano recognizes many of these special cases.

See `local_mul_specialize()`, `local_mul_specialize()`,:func:*local_mul_specialize*

add canonicalization Rearrange expressions of additions and subtractions to a canonical form:

$$(a + b + c + ...) - (z + x + y + ...)$$

See `Canonizer`, `local_add_canonizer`

mul canonicalization Rearrange expressions of multiplication and division to a canonical form:

$$\frac{a * b * c * ...}{z * x * y * ....}$$

See `Canonizer`, `local_mul_canonizer`

dot22 This simple optimization replaces dot(matrix, matrix) with a special *dot22* op that only works for matrix multiplication. This op is implemented with a call to GEMM, and sometimes replaced entirely by the *gemm* optimization.

See `local_dot_to_dot22()`

sparse_dot Theano has a sparse matrix multiplication algorithm that is faster in many cases than scipy's (for dense matrix output). This optimization swaps scipy's algorithm for ours.

See `local_structured_dot()`

sum_scalar_mul This optimizes graphs like `sum(scalar * tensor) -> scalar * sum(tensor)`

See `local_sum_mul_by_scalar()`

neg_neg Composition of two negatives can be cancelled out.

See `local_neg_neg()`

neg_div_neg Matching negatives in both the numerator and denominator can both be removed.

See `local_neg_div_neg()`

add specialization This optimization simplifies expressions involving the addition of zero.

See `local_add_specialize()`

mul specialization Several special cases of mul() exist, and this optimization tries to recognize them. Some examples include: * `mul(x,x)` -> `x**2` * `mul(x,0)` -> `zeros_like(x)` * `mul(x, -1)` -> `neg(x)`

See `local_mul_specialize()`

pow specialization Several special cases of pow() exist, and this optimization tries to recognize them. Some examples include: * `pow(x,2)` -> `x**2` * `pow(x,0)` -> `ones_like(x)` * `pow(x, -0.5)` -> `inv(sqrt(x))`

See `local_pow_specialize()`

inplace_setsubtensor In order to be a pure Op, setsubtensor must copy its entire input, and modify just the subtensor in question (possibly a single element). It is much more efficient to modify that element inplace.

See `local_inplace_setsubtensor()`

gemm Numerical libraries such as MKL and ATLAS implement the BLAS-level-3 interface, and provide a function *GEMM* that implements $Z \leftarrow \alpha A \cdot B + \beta Z$, for matrices *A*, *B* and *Z*, and scalars α, β.

This optimization tries to rearrange a variety of linear algebra expressions into one or more instances of this motif, and replace them each with a single *Gemm* Op.

See `GemmOptimizer`

inplace_elemwise When one of the inputs to an elementwise expression has the same type and shape as the output, and is no longer needed for computation after the elemwise expression is evaluated, then we can reuse the storage of the input to store the output.

See `insert_inplace_optimizer()`

inplace_random Typically when a graph uses random numbers, the RandomState is stored in a shared variable, used once per call and, updated after each function call. In this common case, it makes sense to update the random number generator in-place.

See `random_make_inplace()`

elemwise fusion This optimization compresses subgraphs of computationally cheap elementwise operations into a single Op that does the whole job in a single pass over the inputs (like loop fusion). This is a win when transfer from main memory to the CPU (or from graphics memory to the GPU) is a bottleneck.

See `FusionOptimizer`

GPU transfer The current strategy for choosing which expressions to evaluate on the CPU and which to evaluate on the GPU is a greedy one. There are a number of Ops ***TODO*** with GPU implementations and whenever we find a graph copying data from GPU to CPU in order to evaluate an expression that could have been evaluated on the GPU, we substitute the GPU version of that Op for the CPU version. Likewise if we are copying the output of a Op with a GPU implementation to the GPU, then we substitute the GPU version for the CPU version. In this way, if all goes well, this procedure will result in a graph with the following form:

1. copy non-shared inputs to GPU
2. carry out most/all computations on the GPU
3. copy output back to CPU

When using a GPU, `shared()` will default to GPU storage for 'float32' ndarray arguments, and these shared variables act as seeds for the greedy algorithm.

See `theano.sandbox.cuda.opt.*()`.

local_log_softmax This is a stabilization optimization. It can happen due to rounding errors that the softmax probability of one value gets to 0. Taking the log of 0 would generate -inf that will probably generate NaN later. We return a closer answer.

local_remove_all_assert This is an unsafe optimization. For the fastest possible Theano, this optimization can be enabled by setting `optimizer_including=local_remove_all_assert` which will remove all assertions in the graph for checking user inputs are valid. Use this optimization if you are sure everthing is valid in your graph.

See *unsafe_optimization*

6.2.7 Extending Theano

This advanced tutorial is for users who want to extend Theano with new Types, new Operations (Ops), and new graph optimizations.

Along the way, it also introduces many aspects of how Theano works, so it is also good for you if you are interested in getting more under the hood with Theano itself.

Before tackling this more advanced presentation, it is highly recommended to read the introductory *Tutorial*.

The first few pages will walk you through the definition of a new *Type*, `double`, and a basic arithmetic *operations* on that Type. We will start by defining them using a Python implementation and then we will add a C implementation.

Writing an Op to work on an `ndarray` in C

So suppose you have looked through the library documentation and you don't see a function that does what you want.

If you can implement something in terms of existing Ops, you should do that. Odds are your function that uses existing Theano expressions is short, has no bugs, and potentially profits from optimizations that have already been implemented.

However, if you cannot implement an Op in terms of existing Ops, you have to write a new one. Don't worry, Theano was designed to make it easy to add new Ops, Types, and Optimizations.

This section walks through a non-trivial example Op that does something pretty weird and unrealistic, that is hard to express with existing Ops. (Technically, we could use `Scan` to implement the Op we're about to describe, but we ignore that possibility for the sake of example.)

The following code works, but important error-checking has been omitted for clarity. For example, when you write C code that assumes memory is contiguous, you should check the strides and alignment.

```
class Fibby(theano.Op):
  """
  An arbitrarily generalized Fibbonacci sequence
  """
  __props__ = ()

  def make_node(self, x):
    x_ = tensor.as_tensor_variable(x)
    assert x_.ndim == 1
    return theano.Apply(self,
      inputs=[x_],
      outputs=[x_.type()])
    # using x_.type() is dangerous, it copies x's broadcasting behaviour

  def perform(self, node, inputs, output_storage):
    x, = inputs
    y = output_storage[0][0] = x.copy()
    for i in range(2, len(x)):
      y[i] = y[i-1] * y[i-2] + x[i]

  def c_code(self, node, name, inames, onames, sub):
    x, = inames
    y, = onames
    fail = sub['fail']
    return """
      Py_XDECREF(%(y)s);
      %(y)s = (PyArrayObject*)PyArray_FromArray(
                  %(x)s, 0, NPY_ARRAY_ENSURECOPY);
      if (!%(y)s)
          %(fail)s;
      {//New scope needed to make compilation work
          dtype_%(y)s * y = (dtype_%(y)s*)PyArray_DATA(%(y)s);
          dtype_%(x)s * x = (dtype_%(x)s*)PyArray_DATA(%(x)s);
          for (int i = 2; i < PyArray_DIMS(%(x)s)[0]; ++i)
              y[i] = y[i-1]*y[i-2] + x[i];
      }
    """ % locals()

  def c_code_cache_version(self):
    return (1,)

fibby = Fibby()
```

At a high level, the code fragment declares a class (`Fibby`) and then creates one instance of it (`fibby`). We often gloss over this distinction, but will be precise here: `fibby` (the instance) is an Op, not `Fibby` (the class which is a subclass of `theano.Op`). You can call `fibby(tensor.vector())` on a Variable to build an expression, and in the expression there will be a `.op` attribute that refers to `fibby`.

The first two methods in the Op are relatively boilerplate: `__eq__` and `__hash__`. When two Ops are equal, Theano will merge their outputs if they are applied to the same inputs. The base class (Op) says two objects are equal if (and only if) they are the same object. Writing these boilerplate definitions ensures that

the logic of the equality comparison is always explicit.

It is an essential part of the *Op's contract* that if two Ops compare equal, then they must compute the same result when presented with the same inputs. Here, if we allocated another instance of `Fibby` by typing `fibby2 = Fibby()` then we would have two Ops that behave identically.

When should the implementation of `__eq__` be more complicated? If `Fibby.__init__` had parameters, then we could have configured `fibby2` differently from `fibby` by passing different arguments to the constructor. If we had done that, and if that different configuration made `fibby2` compute different results from `fibby` (for the same inputs) then we would have to add logic to the `__eq__` and `__hash__` function so that he two `Fibby` Ops would *not be equal*. The reason why: Theano's merge optimization looks for Ops comparing equal and merges them. If two Ops compare equal but don't always produce equal results from equal inputs, then you might see wrong calculation.

The `make_node` method creates a node to be included in the expression graph. It runs when we apply our Op (`fibby`) to Variable (`x`), as in `fibby(tensor.vector())`. When an Op has multiple inputs, their order in the inputs argument to `Apply` is important: Theano will call `make_node(*inputs)` to copy the graph, so it is important not to change the semantics of the expression by changing the argument order.

All the `inputs` and `outputs` arguments to `Apply` must be Variables. A common and easy way to ensure inputs are variables is to run them through `as_tensor_variable`. This function leaves TensorType variables alone, raises an error for non-TensorType variables, and copies any `numpy.ndarray` into the storage for a TensorType Constant. The `make_node` method dictates the appropriate Type for all output variables.

The `perform` method implements the Op's mathematical logic in Python. The inputs (here `x`) are passed by value, but a single output is returned indirectly as the first element of single-element lists. If `fibby` had a second output, it would be stored in `output_storage[1][0]`. .. jpt: DOn't understand the following In some execution modes, the output storage might contain the return value of a previous call. That old value can be reused to avoid memory re-allocation, but it must not influence the semantics of the Op output.

The `c_code` method accepts variable names as arguments (`name`, `inames`, `onames`) and returns a C code fragment that computes the expression output. In case of error, the `%(fail)s` statement cleans up and returns properly. The variables `%(x)s` and `%(y)s` are set up by the TensorType to be `PyArrayObject` pointers. TensorType also set up `dtype_%(x)s` to be a typdef to the C type for `x`.

In the first two lines of the C function, we make y point to a new array with the correct size for the output. This is essentially simulating the line `y = x.copy()`.

```
Py_XDECREF(%(y)s);
%(y)s = (PyArrayObject*)PyArray_FromArray(
    %(x)s, 0, NPY_ARRAY_ENSURECOPY);
```

The first line reduces the reference count of the data that y originally pointed to. The second line allocates the new data and makes y point to it.

In C code for a theano op, numpy arrays are represented as `PyArrayObject` C structs. This is part of the numpy/scipy C API documented at http://docs.scipy.org/doc/numpy/reference/c-api.types-and-structures.html

TODO: NEEDS MORE EXPLANATION.

There are some important restrictions to remember when implementing an Op. Unless your Op correctly

defines a `view_map` attribute, the `perform` and `c_code` must not produce outputs whose memory is aliased to any input (technically, if changing the output could change the input object in some sense, they are aliased). Unless your Op correctly defines a `destroy_map` attribute, `perform` and `c_code` must not modify any of the inputs.

TODO: EXPLAIN DESTROYMAP and VIEWMAP BETTER AND GIVE EXAMPLE.

When developing an Op, you should run computations in DebugMode, by using argument `mode='DebugMode'` to `theano.function`. DebugMode is slow, but it can catch many common violations of the Op contract.

TODO: Like what? How? Talk about Python vs. C too.

DebugMode is no silver bullet though. For example, if you modify an Op `self.*` during any of `make_node`, `perform`, or `c_code`, you are probably doing something wrong but DebugMode will not detect this.

TODO: jpt: I don't understand the following sentence.

Ops and Types should usually be considered immutable – you should definitely not make a change that would have an impact on `__eq__`, `__hash__`, or the mathematical value that would be computed by `perform` or `c_code`.

Writing an Optimization

`fibby` of a vector of zeros is another vector of zeros of the same size. Theano does not attempt to infer this from the code provided via `Fibby.perform` or `Fibby.c_code`. However, we can write an optimization that makes use of this observation. This sort of local substitution of special cases is common, and there is a stage of optimization (specialization) devoted to such optimizations. The following optimization (`fibby_of_zero`) tests whether the input is guaranteed to be all zero, and if so it returns the input itself as a replacement for the old output.

TODO: talk about OPTIMIZATION STAGES

```
from theano.tensor.opt import get_scalar_constant_value, NotScalarConstantError

# Remove any fibby(zeros(...))
@theano.tensor.opt.register_specialize
@theano.gof.local_optimizer([fibby])
def fibby_of_zero(node):
  if node.op == fibby:
    x = node.inputs[0]
    try:
      if numpy.all(0 == get_scalar_constant_value(x)):
        return [x]
    except NotScalarConstantError:
      pass
```

The `register_specialize` decorator is what activates our optimization, and tells Theano to use it in the specialization stage. The `local_optimizer` decorator builds a class instance around our global function. The `[fibby]` argument is a hint that our optimizer works on nodes whose `.op` attribute equals `fibby`. The function here (`fibby_of_zero`) expects an `Apply` instance as an argument for parameter

node. It tests using function get_scalar_constant_value, which determines if a Variable (x) is guaranteed to be a constant, and if so, what constant.

Test the optimization

Here is some code to test that the optimization is applied only when needed.

```
# Test it does not apply when not needed
x = T.dvector()
f = function([x], fibby(x))
#theano.printing.debugprint(f)

# We call the function to make sure it runs.
# If you run in DebugMode, it will compare the C and Python outputs.
f(numpy.random.rand(5))
topo = f.maker.fgraph.toposort()
assert len(topo) == 1
assert isinstance(topo[0].op, Fibby)

# Test that the optimization gets applied.
f_zero = function([], fibby(T.zeros([5])))
#theano.printing.debugprint(f_zero)

# If you run in DebugMode, it will compare the output before
# and after the optimization.
f_zero()

# Check that the optimization removes the Fibby Op.
# For security, the Theano memory interface ensures that the output
# of the function is always memory not aliased to the input.
# That is why there is a DeepCopyOp op.
topo = f_zero.maker.fgraph.toposort()
assert len(topo) == 1
assert isinstance(topo[0].op, theano.compile.ops.DeepCopyOp)
```

Overview of the compilation pipeline

The purpose of this page is to explain each step of defining and compiling a Theano function.

Definition of the computation graph

By creating Theano *Variables* using theano.tensor.lscalar or theano.tensor.dmatrix or by using Theano functions such as theano.tensor.sin or theano.tensor.log, the user builds a computation graph. The structure of that graph and details about its components can be found in the *Graph Structures* article.

Compilation of the computation graph

Once the user has built a computation graph, she can use `theano.function` in order to make one or more functions that operate on real data. function takes a list of input *Variables* as well as a list of output Variables that define a precise subgraph corresponding to the function(s) we want to define, compile that subgraph and produce a callable.

Here is an overview of the various steps that are done with the computation graph in the compilation phase:

Step 1 - Create a FunctionGraph The subgraph given by the end user is wrapped in a structure called *FunctionGraph*. That structure defines several hooks on adding and removing (pruning) nodes as well as on modifying links between nodes (for example, modifying an input of an *Apply* node) (see the article about *fgraph – Graph Container [doc TODO]* for more information).

FunctionGraph provides a method to change the input of an Apply node from one Variable to another and a more high-level method to replace a Variable with another. This is the structure that *Optimizers* work on.

Some relevant *Features* are typically added to the FunctionGraph, namely to prevent any optimization from operating inplace on inputs declared as immutable.

Step 2 - Execute main Optimizer Once the FunctionGraph is made, an *optimizer* is produced by the *mode* passed to `function` (the Mode basically has two important fields, `linker` and `optimizer`). That optimizer is applied on the FunctionGraph using its optimize() method.

The optimizer is typically obtained through `optdb`.

Step 3 - Execute linker to obtain a thunk Once the computation graph is optimized, the *linker* is extracted from the Mode. It is then called with the FunctionGraph as argument to produce a `thunk`, which is a function with no arguments that returns nothing. Along with the thunk, one list of input containers (a theano.gof.Container is a sort of object that wraps another and does type casting) and one list of output containers are produced, corresponding to the input and output Variables as well as the updates defined for the inputs when applicable. To perform the computations, the inputs must be placed in the input containers, the thunk must be called, and the outputs must be retrieved from the output containers where the thunk put them.

Typically, the linker calls the `toposort` method in order to obtain a linear sequence of operations to perform. How they are linked together depends on the Linker used. The CLinker produces a single block of C code for the whole computation, whereas the OpWiseCLinker produces one thunk for each individual operation and calls them in sequence.

The linker is where some options take effect: the `strict` flag of an input makes the associated input container do type checking. The `borrow` flag of an output, if False, adds the output to a `no_recycling` list, meaning that when the thunk is called the output containers will be cleared (if they stay there, as would be the case if `borrow` was True, the thunk would be allowed to reuse (or "recycle") the storage).

Note: Compiled libraries are stored within a specific compilation directory, which by default is set to `$HOME/.theano/compiledir_xxx`, where `xxx` identifies the platform (under Windows the default

location is instead `$LOCALAPPDATA\Theano\compiledir_xxx`). It may be manually set to a different location either by setting `config.compiledir` or `config.base_compiledir`, either within your Python script or by using one of the configuration mechanisms described in `config`.

The compile cache is based upon the C++ code of the graph to be compiled. So, if you change compilation configuration variables, such as `config.blas.ldflags`, you will need to manually remove your compile cache, using `Theano/bin/theano-cache clear`

Theano also implements a lock mechanism that prevents multiple compilations within the same compilation directory (to avoid crashes with paralell execution of some scripts). This mechanism is currently enabled by default, but if it causes any problem it may be disabled using the function `theano.gof.compilelock.set_lock_status(..)`.

Step 4 - Wrap the thunk in a pretty package The thunk returned by the linker along with input and output containers is unwieldy. `function` hides that complexity away so that it can be used like a normal function with arguments and return values.

Theano vs. C

We describe some of the patterns in Theano, and present their closest analogue in a statically typed language such as C:

Theano	C
Apply	function application / function call
Variable	local function data / variable
Shared Variable	global function data / variable
Op	operations carried out in computation / function definition
Type	data types

For example:

```
int d = 0;

int main(int a) {
    int b = 3;
    int c = f(b)
    d = b + c;
    return g(a, c);
}
```

Based on this code snippet, we can relate `f` and `g` to Ops, `a`, `b` and `c` to Variables, `d` to Shared Variable, `g(a, c)`, `f(b)` and `d = b + c` (taken as meaning the action of computing `f`, `g` or `+``on their respective inputs) to Applies. Lastly, ``int` could be interpreted as the Theano Type of the Variables `a`, `b`, `c` and `d`.

Graph Structures

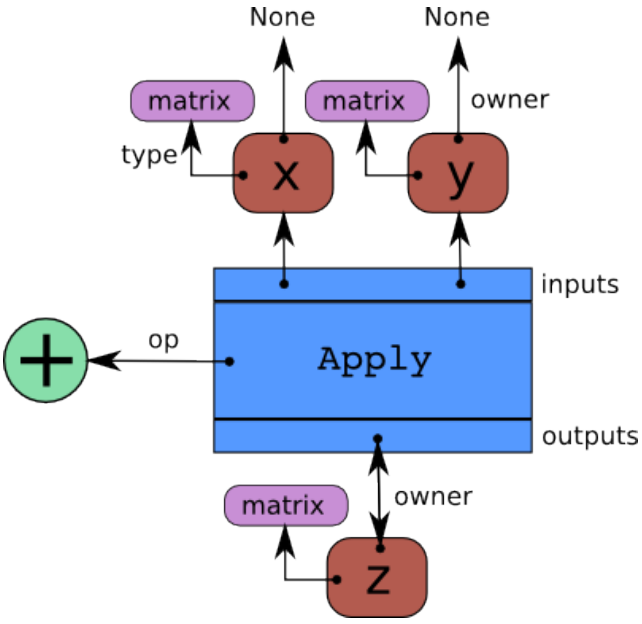
Theano represents symbolic mathematical computations as graphs. These graphs are composed of interconnected *Apply* and *Variable* nodes. They are associated to *function application* and *data*, respectively.

Operations are represented by *Op* instances and data types are represented by *Type* instances. Here is a piece of code and a diagram showing the structure built by that piece of code. This should help you understand how these pieces fit together:

Code

```
x = dmatrix('x')
y = dmatrix('y')
z = x + y
```

Diagram



Arrows represent references to the Python objects pointed at. The blue box is an *Apply* node. Red boxes are *Variable* nodes. Green circles are *Ops*. Purple boxes are *Types*.

When we create *Variables* and then *Apply Ops* to them to make more Variables, we build a bi-partite, directed, acyclic graph. Variables point to the Apply nodes representing the function application producing them via their `owner` field. These Apply nodes point in turn to their input and output Variables via their `inputs` and `outputs` fields. (Apply instances also contain a list of references to their `outputs`, but those pointers don't count in this graph.)

The `owner` field of both `x` and `y` point to `None` because they are not the result of another computation. If one of them was the result of another computation, it's `owner` field would point to another blue box like `z` does, and so on.

Note that the `Apply` instance's outputs points to `z`, and `z.owner` points back to the `Apply` instance.

An explicit example

In this example we will compare two ways of defining the same graph. First, a short bit of code will build an expression (graph) the *normal* way, with most of the graph construction being done automatically. Second,

we will walk through a longer re-coding of the same thing without any shortcuts, that will make the graph construction very explicit.

Short example

This is what you would normally type:

```
# create 3 Variables with owner = None
x = T.matrix('x')
y = T.matrix('y')
z = T.matrix('z')

# create 2 Variables (one for 'e', one intermediate for y*z)
# create 2 Apply instances (one for '+', one for '*')
e = x + y * z
```

Long example

This is what you would type to build the graph explicitly:

```
from theano.tensor import add, mul, Apply, Variable, TensorType

# Instantiate a type that represents a matrix of doubles
float64_matrix = TensorType(dtype = 'float64',              # double
                            broadcastable = (False, False)) # matrix

# We make the Variable instances we need.
x = Variable(type = float64_matrix, name = 'x')
y = Variable(type = float64_matrix, name = 'y')
z = Variable(type = float64_matrix, name = 'z')

# This is the Variable that we want to symbolically represents y*z
mul_variable = Variable(type = float64_matrix)
assert mul_variable.owner is None

# Instantiate a symbolic multiplication
node_mul = Apply(op = mul,
                 inputs = [y, z],
                 outputs = [mul_variable])
# Fields 'owner' and 'index' are set by Apply
assert mul_variable.owner is node_mul
# 'index' is the position of mul_variable in mode_mul's outputs
assert mul_variable.index == 0

# This is the Variable that we want to symbolically represents x+(y*z)
add_variable = Variable(type = float64_matrix)
assert add_variable.owner is None

# Instantiate a symbolic addition
node_add = Apply(op = add,
                 inputs = [x, mul_variable],
                 outputs = [add_variable])
# Fields 'owner' and 'index' are set by Apply
assert add_variable.owner is node_add
assert add_variable.index == 0
```

```
e = add_variable

# We have access to x, y and z through pointers
assert e.owner.inputs[0] is x
assert e.owner.inputs[1] is mul_variable
assert e.owner.inputs[1].owner.inputs[0] is y
assert e.owner.inputs[1].owner.inputs[1] is z
```

Note how the call to `Apply` modifies the `owner` and `index` fields of the *Variables* passed as outputs to point to itself and the rank they occupy in the output list. This whole machinery builds a DAG (Directed Acyclic Graph) representing the computation, a graph that Theano can compile and optimize.

Automatic wrapping All nodes in the graph must be instances of `Apply` or `Result`, but `<Op subclass>.make_node()` typically wraps constants to satisfy those constraints. For example, the `tensor.add()` Op instance is written so that:

```
e = dscalar('x') + 1
```

builds the following graph:

```
node = Apply(op = add,
             inputs = [Variable(type = dscalar, name = 'x'),
                       Constant(type = lscalar, data = 1)],
             outputs = [Variable(type = dscalar)])
e = node.outputs[0]
```

Graph Structures

The following section outlines each type of structure that may be used in a Theano-built computation graph. The following structures are explained: *Apply*, *Constant*, *Op*, *Variable* and *Type*.

Apply An *Apply node* is a type of internal node used to represent a *computation graph* in Theano. Unlike *Variable nodes*, Apply nodes are usually not manipulated directly by the end user. They may be accessed via a Variable's `owner` field.

An Apply node is typically an instance of the `Apply` class. It represents the application of an *Op* on one or more inputs, where each input is a *Variable*. By convention, each Op is responsible for knowing how to build an Apply node from a list of inputs. Therefore, an Apply node may be obtained from an Op and a list of inputs by calling `Op.make_node(*inputs)`.

Comparing with the Python language, an *Apply* node is Theano's version of a function call whereas an *Op* is Theano's version of a function definition.

An Apply instance has three important fields:

op An *Op* that determines the function/transformation being applied here.

inputs A list of *Variables* that represent the arguments of the function.

outputs A list of *Variables* that represent the return values of the function.

An Apply instance can be created by calling `gof.Apply(op, inputs, outputs)`.

Op An *Op* in Theano defines a certain computation on some types of inputs, producing some types of outputs. It is equivalent to a function definition in most programming languages. From a list of input *Variables* and an Op, you can build an *Apply* node representing the application of the Op to the inputs.

It is important to understand the distinction between an Op (the definition of a function) and an Apply node (the application of a function). If you were to interpret the Python language using Theano's structures, code going like `def f(x): ...` would produce an Op for `f` whereas code like `a = f(x)` or `g(f(4), 5)` would produce an Apply node involving the `f` Op.

Type A *Type* in Theano represents a set of constraints on potential data objects. These constraints allow Theano to tailor C code to handle them and to statically optimize the computation graph. For instance, the *irow* type in the `theano.tensor` package gives the following constraints on the data the Variables of type `irow` may contain:

1. Must be an instance of `numpy.ndarray`: `isinstance(x, numpy.ndarray)`
2. Must be an array of 32-bit integers: `str(x.dtype) == 'int32'`
3. Must have a shape of 1xN: `len(x.shape) == 2 and x.shape[0] == 1`

Knowing these restrictions, Theano may generate C code for addition, etc. that declares the right data types and that contains the right number of loops over the dimensions.

Note that a Theano *Type* is not equivalent to a Python type or class. Indeed, in Theano, *irow* and *dmatrix* both use `numpy.ndarray` as the underlying type for doing computations and storing data, yet they are different Theano Types. Indeed, the constraints set by `dmatrix` are:

1. Must be an instance of `numpy.ndarray`: `isinstance(x, numpy.ndarray)`
2. Must be an array of 64-bit floating point numbers: `str(x.dtype) == 'float64'`
3. Must have a shape of MxN, no restriction on M or N: `len(x.shape) == 2`

These restrictions are different from those of `irow` which are listed above.

There are cases in which a Type can fully correspond to a Python type, such as the `double` Type we will define here, which corresponds to Python's `float`. But, it's good to know that this is not necessarily the case. Unless specified otherwise, when we say "Type" we mean a Theano Type.

Variable A *Variable* is the main data structure you work with when using Theano. The symbolic inputs that you operate on are Variables and what you get from applying various Ops to these inputs are also Variables. For example, when I type

```
>>> x = theano.tensor.ivector()
>>> y = -x
```

`x` and `y` are both Variables, i.e. instances of the `Variable` class. The *Type* of both `x` and `y` is `theano.tensor.ivector`.

Unlike `x`, `y` is a Variable produced by a computation (in this case, it is the negation of `x`). `y` is the Variable corresponding to the output of the computation, while `x` is the Variable corresponding to its input. The

computation itself is represented by another type of node, an *Apply* node, and may be accessed through `y.owner`.

More specifically, a Variable is a basic structure in Theano that represents a datum at a certain point in computation. It is typically an instance of the class `Variable` or one of its subclasses.

A Variable `r` contains four important fields:

type a *Type* defining the kind of value this Variable can hold in computation.

owner this is either None or an *Apply* node of which the Variable is an output.

index the integer such that `owner.outputs[index] is r` (ignored if `owner` is None)

name a string to use in pretty-printing and debugging.

Variable has one special subclass: *Constant*.

Constant A Constant is a *Variable* with one extra field, *data* (only settable once). When used in a computation graph as the input of an *Op application*, it is assumed that said input will *always* take the value contained in the constant's data field. Furthermore, it is assumed that the *Op* will not under any circumstances modify the input. This means that a constant is eligible to participate in numerous optimizations: constant inlining in C code, constant folding, etc.

A constant does not need to be specified in a `function`'s list of inputs. In fact, doing so will raise an exception.

Graph Structures Extension

When we start the compilation of a Theano function, we compute some extra information. This section describes a portion of the information that is made available. Not everything is described, so email theano-dev if you need something that is missing.

The graph gets cloned at the start of compilation, so modifications done during compilation won't affect the user graph.

Each variable receives a new field called clients. It is a list with references to every place in the graph where this variable is used. If its length is 0, it means the variable isn't used. Each place where it is used is described by a tuple of 2 elements. There are two types of pairs:

- The first element is an Apply node.
- The first element is the string "output". It means the function outputs this variable.

In both types of pairs, the second element of the tuple is an index, such that: `var.clients[*][0].inputs[index]` or `fgraph.outputs[index]` is that variable.

```
import theano
v = theano.tensor.vector()
f = theano.function([v], (v+1).sum())
theano.printing.debugprint(f)
# Sorted list of all nodes in the compiled graph.
topo = f.maker.fgraph.toposort()
topo[0].outputs[0].clients
```

```
# [(Sum(Elemwise{add,no_inplace}.0), 0)]
topo[1].outputs[0].clients
# [('output', 0)]

# An internal variable
var = topo[0].outputs[0]
client = var.clients[0]
client
# (Sum(Elemwise{add,no_inplace}.0), 0)
type(client[0])
# <class 'theano.gof.graph.Apply'>
assert client[0].inputs[client[1]] is var

# An output of the graph
var = topo[1].outputs[0]
client = var.clients[0]
client
# ('output', 0)
assert f.maker.fgraph.outputs[client[1]] is var
```

Making the double type

Type's contract

In Theano's framework, a `Type` (`gof.type.Type`) is any object which defines the following methods. To obtain the default methods described below, the Type should be an instance of `Type` or should be an instance of a subclass of `Type`. If you will write all methods yourself, you need not use an instance of `Type`.

Methods with default arguments must be defined with the same signature, i.e. the same default argument names and values. If you wish to add extra arguments to any of these methods, these extra arguments must have default values.

class `PureType`

`filter`(*value*, *strict=False*, *allow_downcast=None*)

This casts a value to match the Type and returns the cast value. If `value` is incompatible with the Type, the method must raise an exception. If `strict` is True, `filter` must return a reference to `value` (i.e. casting prohibited). If `strict` is False, then casting may happen, but downcasting should only be used in two situations:

- if `allow_downcast` is True
- if `allow_downcast` is `None` and the default behavior for this type allows downcasting for the given `value` (this behavior is type-dependent, you may decide what your own type does by default)

We need to define `filter` with three arguments. The second argument must be called `strict` (Theano often calls it by keyword) and must have a default value of `False`. The third argument must be called `allow_downcast` and must have a default value of `None`.

filter_inplace(*value*, *storage*, *strict=False*, *allow_downcast=None*)
If filter_inplace is defined, it will be called instead of filter() This is to allow reusing the old allocated memory. As of this writing this is used only when we transfer new data to a shared variable on the gpu.

`storage` will be the old value. i.e. The old numpy array, CudaNdarray, ...

is_valid_value(*value*)
Returns True iff the value is compatible with the Type. If `filter(value, strict = True)` does not raise an exception, the value is compatible with the Type.

Default: True iff `filter(value, strict=True)` does not raise an exception.

values_eq(*a*, *b*)
Returns True iff `a` and `b` are equal.

Default: `a == b`

values_eq_approx(*a*, *b*)
Returns True iff `a` and `b` are approximately equal, for a definition of "approximately" which varies from Type to Type.

Default: `values_eq(a, b)`

make_variable(*name=None*)
Makes a *Variable* of this Type with the specified name, if `name` is not `None`. If `name` is `None`, then the Variable does not have a name. The Variable will have its `type` field set to the Type object.

Default: there is a generic definition of this in Type. The Variable's `type` will be the object that defines this method (in other words, `self`).

__call__(*name=None*)
Syntactic shortcut to `make_variable`.

Default: `make_variable`

__eq__(*other*)
Used to compare Type instances themselves

Default: `object.__eq__`

__hash__()
Types should not be mutable, so it should be OK to define a hash function. Typically this function should hash all of the terms involved in `__eq__`.

Default: `id(self)`

get_shape_info(*obj*)
Optional. Only needed to profile the memory of this Type of object.

Return the information needed to compute the memory size of `obj`.

The memory size is only the data, so this excludes the container. For an ndarray, this is the data, but not the ndarray object and other data structures such as shape and strides.

`get_shape_info()` and `get_size()` work in tandem for the memory profiler.

`get_shape_info()` is called during the execution of the function. So it is better that it is not too slow.

`get_size()` will be called on the output of this function when printing the memory profile.

Parameters **obj** – The object that this Type represents during execution

Returns Python object that `self.get_size()` understands

get_size(*shape_info*)
Number of bytes taken by the object represented by shape_info.

Optional. Only needed to profile the memory of this Type of object.

Parameters **shape_info** – the output of the call to get_shape_info()

Returns the number of bytes taken by the object described by `shape_info`.

clone(*dtype=None*, *broadcastable=None*)
Optional, for TensorType-alikes.

Return a copy of the type with a possibly changed value for dtype and broadcastable (if they aren't *None*).

Parameters

- **dtype** – New dtype for the copy.
- **broadcastable** – New broadcastable tuple for the copy.

may_share_memory(*a*, *b*)
Optional to run, but mandatory for DebugMode. Return True if the Python objects *a* and *b* could share memory. Return False otherwise. It is used to debug when Ops did not declare memory aliasing between variables. Can be a static method. It is highly recommended to use and is mandatory for Type in Theano as our buildbot runs in DebugMode.

For each method, the *default* is what `Type` defines for you. So, if you create an instance of `Type` or an instance of a subclass of `Type`, you must define `filter`. You might want to override `values_eq_approx`, as well as `values_eq`. The other defaults generally need not be overridden.

For more details you can go see the documentation for *Type*.

Defining double

We are going to base Type `double` on Python's `float`. We must define `filter` and shall override `values_eq_approx`.

filter

```
# Note that we shadow Python's function ''filter'' with this
# definition.
def filter(x, strict=False, allow_downcast=None):
    if strict:
        if isinstance(x, float):
            return x
        else:
```

```
            raise TypeError('Expected a float!')
    elif allow_downcast:
        return float(x)
    else:   # Covers both the False and None cases.
        x_float = float(x)
        if x_float == x:
            return x_float
        else:
             raise TypeError('The double type cannot accurately represent '
                             'value %s (of type %s): you must explicitly '
                             'allow downcasting if you want to do this.'
                             % (x, type(x)))
```

If `strict` is True we need to return `x`. If `strict` is True and `x` is not a `float` (for example, `x` could easily be an `int`) then it is incompatible with our Type and we must raise an exception.

If `strict is False` then we are allowed to cast `x` to a `float`, so if `x` is an `int` it we will return an equivalent `float`. However if this cast triggers a precision loss (`x != float(x)`) and `allow_downcast` is not True, then we also raise an exception. Note that here we decided that the default behavior of our type (when `allow_downcast` is set to `None`) would be the same as when `allow_downcast` is False, i.e. no precision loss is allowed.

values_eq_approx

```
def values_eq_approx(x, y, tolerance=1e-4):
    return abs(x - y) / (abs(x) + abs(y)) < tolerance
```

The second method we define is `values_eq_approx`. This method allows approximate comparison between two values respecting our Type's constraints. It might happen that an optimization changes the computation graph in such a way that it produces slightly different variables, for example because of numerical instability like rounding errors at the end of the mantissa. For instance, `a + a + a + a + a + a` might not actually produce the exact same output as `6 * a` (try with a=0.1), but with `values_eq_approx` we do not necessarily mind.

We added an extra `tolerance` argument here. Since this argument is not part of the API, it must have a default value, which we chose to be 1e-4.

Note: `values_eq` is never actually used by Theano, but it might be used internally in the future. Equality testing in *DebugMode* is done using `values_eq_approx`.

Putting them together

What we want is an object that respects the aforementioned contract. Recall that Type defines default implementations for all required methods of the interface, except `filter`. One way to make the Type is to instantiate a plain Type and set the needed fields:

```
from theano import gof

double = gof.Type()
double.filter = filter
double.values_eq_approx = values_eq_approx
```

Another way to make this Type is to make a subclass of `gof.Type` and define `filter` and `values_eq_approx` in the subclass:

```
from theano import gof

class Double(gof.Type):

    def filter(self, x, strict=False, allow_downcast=None):
        # See code above.
        ...

    def values_eq_approx(self, x, y, tolerance=1e-4):
        # See code above.
        ...

double = Double()
```

`double` is then an instance of Type `Double`, which in turn is a subclass of `Type`.

There is a small issue with defining `double` this way. All instances of `Double` are technically the same Type. However, different `Double` Type instances do not compare the same:

```
>>> double1 = Double()
>>> double2 = Double()
>>> double1 == double2
False
```

Theano compares Types using == to see if they are the same. This happens in DebugMode. Also, Ops can (and should) ensure that their inputs have the expected Type by checking something like `if x.type == lvector`.

There are several ways to make sure that equality testing works properly:

1. Define `Double.__eq__` so that instances of type Double are equal. For example:

   ```
   def __eq__(self, other):
       return type(self) is Double and type(other) is Double
   ```

2. Override `Double.__new__` to always return the same instance.
3. Hide the Double class and only advertise a single instance of it.

Here we will prefer the final option, because it is the simplest. Ops in the Theano code often define the `__eq__` method though.

Untangling some concepts

Initially, confusion is common on what an instance of Type is versus a subclass of Type or an instance of Variable. Some of this confusion is syntactic. A Type is any object which has fields corresponding to the functions defined above. The Type class provides sensible defaults for all of them except `filter`, so when defining new Types it is natural to subclass Type. Therefore, we often end up with Type subclasses and it is can be confusing what these represent semantically. Here is an attempt to clear up the confusion:

- An **instance of Type** (or an instance of a subclass) is a set of constraints on real data. It is akin to a primitive type or class in C. It is a *static* annotation.
- An **instance of Variable** symbolizes data nodes in a data flow graph. If you were to parse the C expression `int x;`, `int` would be a Type instance and `x` would be a Variable instance of that Type instance. If you were to parse the C expression `c = a + b;`, `a`, `b` and `c` would all be Variable instances.
- A **subclass of Type** is a way of implementing a set of Type instances that share structural similarities. In the `double` example that we are doing, there is actually only one Type in that set, therefore the subclass does not represent anything that one of its instances does not. In this case it is a singleton, a set with one element. However, the `TensorType` class in Theano (which is a subclass of Type) represents a set of types of tensors parametrized by their data type or number of dimensions. We could say that subclassing Type builds a hierarchy of Types which is based upon structural similarity rather than compatibility.

Final version

```
from theano import gof

class Double(gof.Type):

    def filter(self, x, strict=False, allow_downcast=None):
        if strict:
            if isinstance(x, float):
                return x
            else:
                raise TypeError('Expected a float!')
        elif allow_downcast:
            return float(x)
        else:    # Covers both the False and None cases.
            x_float = float(x)
            if x_float == x:
                return x_float
            else:
                 raise TypeError('The double type cannot accurately represent '
                                 'value %s (of type %s): you must explicitly '
                                 'allow downcasting if you want to do this.'
                                 % (x, type(x)))

    def values_eq_approx(self, x, y, tolerance=1e-4):
        return abs(x - y) / (abs(x) + abs(y)) < tolerance

    def __str__(self):
        return "double"

double = Double()
```

We add one utility function, `__str__`. That way, when we print `double`, it will print out something intelligible.

Making arithmetic Ops on double

Now that we have a `double` type, we have yet to use it to perform computations. We'll start by defining multiplication.

Op's contract

An Op is any object which inherits from `gof.Op`. It has to define the following methods.

`make_node`(**inputs*)
This method is responsible for creating output Variables of a suitable symbolic Type to serve as the outputs of this Op's application. The Variables found in `*inputs` must be operated on using Theano's symbolic language to compute the symbolic output Variables. This method should put these outputs into an Apply instance, and return the Apply instance.

This method creates an Apply node representing the application of the Op on the inputs provided. If the Op cannot be applied to these inputs, it must raise an appropriate exception.

The inputs of the Apply instance returned by this call must be ordered correctly: a subsequent `self.make_node(*apply.inputs)` must produce something equivalent to the first `apply`.

`perform`(*node*, *inputs*, *output_storage*)
This method computes the function associated to this Op. `node` is an Apply node created by the Op's `make_node` method. `inputs` is a list of references to data to operate on using non-symbolic statements, (i.e., statements in Python, Numpy). `output_storage` is a list of storage cells where the variables of the computation must be put.

More specifically:

- `node`: This is a reference to an Apply node which was previously obtained via the `Op`'s `make_node` method. It is typically not used in simple Ops, but it contains symbolic information that could be required for complex Ops.
- `inputs`: This is a list of data from which the values stored in `output_storage` are to be computed using non-symbolic language.
- `output_storage`: This is a list of storage cells where the output is to be stored. A storage cell is a one-element list. It is forbidden to change the length of the list(s) contained in `output_storage`. There is one storage cell for each output of the Op.

  The data put in `output_storage` must match the type of the symbolic output. This is a situation where the `node` argument can come in handy.

  A function Mode may allow `output_storage` elements to persist between evaluations, or it may reset `output_storage` cells to hold a value of `None`. It can also pre-allocate some memory for the Op to use. This feature can allow `perform` to reuse memory between calls, for example. If there is something preallocated in the `output_storage`, it will be of the good dtype, but can have the wrong shape and have any stride pattern.

This method must be determined by the inputs. That is to say, if it is evaluated once on inputs A and returned B, then if ever inputs C, equal to A, are presented again, then outputs equal to B must be returned again.

You must be careful about aliasing outputs to inputs, and making modifications to any of the inputs. See *Views and inplace operations* before writing a `perform` implementation that does either of these things.

Instead (or in addition to) `perform()` You can also provide a *C implementation* of For more details, refer to the documentation for *Op*.

__eq__(*other*)
`other` is also an Op.

Returning `True` here is a promise to the optimization system that the other Op will produce exactly the same graph effects (from perform) as this one, given identical inputs. This means it will produce the same output values, it will destroy the same inputs (same destroy_map), and will alias outputs to the same inputs (same view_map). For more details, see *Views and inplace operations*.

Note: If you set *__props__*, this will be automatically generated.

__hash__()
If two Op instances compare equal, then they **must** return the same hash value.

Equally important, this hash value must not change during the lifetime of self. Op instances should be immutable in this sense.

Note: If you set *__props__*, this will be automatically generated.

Optional methods or attributes

__props__
Default: Undefined

Must be a tuple. Lists the name of the attributes which influence the computation performed. This will also enable the automatic generation of appropriate __eq__, __hash__ and __str__ methods. Should be set to *()* if you have no attributes that are relevant to the computation to generate the methods.

New in version 0.7.

default_output
Default: None

If this member variable is an integer, then the default implementation of `__call__` will return `node.outputs[self.default_output]`, where `node` was returned by `make_node`. Otherwise, the entire list of outputs will be returned, unless it is of length 1, where the single element will be returned by itself.

make_thunk(*node*, *storage_map*, *compute_map*, *no_recycling*)
This function must return a thunk, that is a zero-arguments function that encapsulates the computation to be performed by this op on the arguments of the node.

Parameters

- **node** – Apply instance The node for which a thunk is requested.

- **storage_map** – dict of lists This maps variables to a one-element lists holding the variable's current value. The one-element list acts as pointer to the value and allows sharing that "pointer" with other nodes and instances.
- **compute_map** – dict of lists This maps variables to one-element lists holding booleans. If the value is 0 then the variable has not been computed and the value should not be considered valid. If the value is 1 the variable has been computed and the value is valid. If the value is 2 the variable has been garbage-collected and is no longer valid, but shouldn't be required anymore for this call.
- **no_recycling** – WRITEME WRITEME

The returned function must ensure that is sets the computed variables as computed in the *compute_map*.

Defining this function removes the requirement for `perform()` or C code, as you will define the thunk for the computation yourself.

__call__(**inputs*, ***kwargs*)

By default this is a convenience function which calls `make_node()` with the supplied arguments and returns the result indexed by *default_output*. This can be overridden by subclasses to do anything else, but must return either a theano Variable or a list of Variables.

If you feel the need to override *__call__* to change the graph based on the arguments, you should instead create a function that will use your Op and build the graphs that you want and call that instead of the Op instance directly.

infer_shape(*node*, *shapes*)

This function is needed for shape optimization. `shapes` is a list with one tuple for each input of the Apply node (which corresponds to the inputs of the op). Each tuple contains as many elements as the number of dimensions of the corresponding input. The value of each element is the shape (number of items) along the corresponding dimension of that specific input.

While this might sound complicated, it is nothing more than the shape of each input as symbolic variables (one per dimension).

The function should return a list with one tuple for each output. Each tuple should contain the corresponding output's computed shape.

Implementing this method will allow Theano to compute the output's shape without computing the output itself, potentially sparing you a costly recomputation.

flops(*inputs*, *outputs*)

It is only used to have more information printed by the memory profiler. It makes it print the mega flops and giga flops per second for each apply node. It takes as inputs two lists: one for the inputs and one for the outputs. They contain tuples that are the shapes of the corresponding inputs/outputs.

__str__()

This allows you to specify a more informative string representation of your Op. If an Op has parameters, it is highly recommended to have the `__str__` method include the name of the op and the Op's parameters' values.

Note: If you set *__props__*, this will be automatically generated. You can still overide it for custom output.

do_constant_folding(*node*)

Default: Return True

By default when optimizations are enabled, we remove during function compilation Apply nodes whose inputs are all constants. We replace the Apply node with a Theano constant variable. This way, the Apply node is not executed at each function call. If you want to force the execution of an op during the function call, make do_constant_folding return False.

As done in the Alloc op, you can return False only in some cases by analyzing the graph from the node parameter.

If you want your op to work with gradient.grad() you also need to implement the functions described below.

Gradient

These are the function required to work with gradient.grad().

grad(*inputs*, *output_gradients*)

If the Op being defined is differentiable, its gradient may be specified symbolically in this method. Both `inputs` and `output_gradients` are lists of symbolic Theano Variables and those must be operated on using Theano's symbolic language. The grad method must return a list containing one Variable for each input. Each returned Variable represents the gradient with respect to that input computed based on the symbolic gradients with respect to each output.

If the output is not differentiable with respect to an input then this method should be defined to return a variable of type NullType for that input. Likewise, if you have not implemented the grad computation for some input, you may return a variable of type NullType for that input. theano.gradient contains convenience methods that can construct the variable for you: `theano.gradient.grad_undefined()` and `theano.gradient.grad_not_implemented()`, respectively.

If an element of output_gradient is of type theano.gradient.DisconnectedType, it means that the cost is not a function of this output. If any of the op's inputs participate in the computation of only disconnected outputs, then Op.grad should return DisconnectedType variables for those inputs.

If the grad method is not defined, then Theano assumes it has been forgotten. Symbolic differentiation will fail on a graph that includes this Op.

It must be understood that the Op's grad method is not meant to return the gradient of the Op's output. theano.tensor.grad computes gradients; Op.grad is a helper function that computes terms that appear in gradients.

If an Op has a single vector-valued output y and a single vector-valued input x, then the grad method will be passed x and a second vector z. Define J to be the Jacobian of y with respect to x. The Op's grad method should return dot(J.T,z). When theano.tensor.grad calls the grad method, it will set z to be the gradient of the cost C with respect to y. If this op is the only op that acts on x, then dot(J.T,z) is the gradient of C with respect to x. If there are other ops that act on x, theano.tensor.grad will have to add up the terms of x's gradient contributed by the other op's grad method.

In practice, an op's input and output are rarely implemented as single vectors. Even if an op's output consists of a list containing a scalar, a sparse matrix, and a 4D tensor, you can think of these objects as being formed by rearranging a vector. Likewise for the input. In this view, the values computed by the grad method still represent a Jacobian-vector product.

In practice, it is probably not a good idea to explicitly construct the Jacobian, which might be very large and very sparse. However, the returned value should be equal to the Jacobian-vector product.

So long as you implement this product correctly, you need not understand what theano.tensor.grad is doing, but for the curious the mathematical justification is as follows:

In essence, the grad method must simply implement through symbolic Variables and operations the chain rule of differential calculus. The chain rule is the mathematical procedure that allows one to calculate the total derivative $\frac{dC}{dx}$ of the final scalar symbolic Variable C with respect to a primitive symbolic Variable x found in the list `inputs`. The grad method does this using `output_gradients` which provides the total derivative $\frac{dC}{df}$ of C with respect to a symbolic Variable that is returned by the Op (this is provided in `output_gradients`), as well as the knowledge of the total derivative $\frac{df}{dx}$ of the latter with respect to the primitive Variable (this has to be computed).

In mathematics, the total derivative of a scalar variable (C) with respect to a vector of scalar variables (x), i.e. the gradient, is customarily represented as the row vector of the partial derivatives, whereas the total derivative of a vector of scalar variables (f) with respect to another (x), is customarily represented by the matrix of the partial derivatives, i.e.the jacobian matrix. In this convenient setting, the chain rule instructs that the gradient of the final scalar variable C with respect to the primitive scalar variables in x through those in f is simply given by the matrix product: $\frac{dC}{dx} = \frac{dC}{df} * \frac{df}{dx}$.

Here, the chain rule must be implemented in a similar but slightly more complex setting: Theano provides in the list `output_gradients` one gradient for each of the Variables returned by the Op. Where f is one such particular Variable, the corresponding gradient found in `output_gradients` and representing $\frac{dC}{df}$ is provided with a shape similar to f and thus not necessarily as a row vector of scalars. Furthermore, for each Variable x of the Op's list of input variables `inputs`, the returned gradient representing $\frac{dC}{dx}$ must have a shape similar to that of Variable x.

If the output list of the op is $[f_1, ...f_n]$, then the list `output_gradients` is $[grad_{f_1}(C), grad_{f_2}(C), ..., grad_{f_n}(C)]$. If `inputs` consists of the list $[x_1, ..., x_m]$, then Op.grad should return the list $[grad_{x_1}(C), grad_{x_2}(C), ..., grad_{x_m}(C)]$, where $(grad_y(Z))_i = \frac{\partial Z}{\partial y_i}$ (and i can stand for multiple dimensions).

In other words, `grad()` does not return $\frac{df_i}{dx_j}$, but instead the appropriate dot product specified by the chain rule: $\frac{dC}{dx_j} = \frac{dC}{df_i} \cdot \frac{df_i}{dx_j}$. Both the partial differentiation and the multiplication have to be performed by `grad()`.

Theano currently imposes the following constraints on the values returned by the grad method:

1.They must be Variable instances.

2.When they are types that have dtypes, they must never have an integer dtype.

The output gradients passed *to* Op.grad will also obey these constraints.

Integers are a tricky subject. Integers are the main reason for having DisconnectedType, NullType or zero gradient. When you have an integer as an argument to your grad method, recall the definition of a derivative to help you decide what value to return:

$\frac{df}{dx} = \lim_{\epsilon \to 0}(f(x+\epsilon) - f(x))/\epsilon$.

Suppose your function f has an integer-valued output. For most functions you're likely to implement in theano, this means your gradient should be zero, because f(x+epsilon) = f(x) for almost all x. (The only other option is that the gradient could be undefined, if your function is discontinuous everywhere, like the rational indicator function)

Suppose your function f has an integer-valued input. This is a little trickier, because you need to think about what you mean mathematically when you make a variable integer-valued in theano. Most of the time in machine learning we mean "f is a function of a real-valued x, but we are only going to pass in integer-values of x". In this case, f(x+epsilon) exists, so the gradient through f should be the same whether x is an integer or a floating point variable. Sometimes what we mean is "f is a function of an integer-valued x, and f is only defined where x is an integer." Since f(x+epsilon) doesn't exist, the gradient is undefined. Finally, many times in theano, integer valued inputs don't actually affect the elements of the output, only its shape.

If your function f has both an integer-valued input and an integer-valued output, then both rules have to be combined:

- If f is defined at (x+epsilon), then the input gradient is defined. Since f(x+epsilon) would be equal to f(x) almost everywhere, the gradient should be 0 (first rule).
- If f is only defined where x is an integer, then the gradient is undefined, regardless of what the gradient with respect to the output is.

Examples:

1. **f(x,y) = dot product between x and y. x and y are integers.** Since the output is also an integer, f is a step function. Its gradient is zero almost everywhere, so Op.grad should return zeros in the shape of x and y.
2. **f(x,y) = dot product between x and y. x is floating point and y is an integer.** In this case the output is floating point. It doesn't matter that y is an integer. We consider f to still be defined at f(x,y+epsilon). The gradient is exactly the same as if y were floating point.
3. **f(x,y) = argmax of x along axis y.** The gradient with respect to y is undefined, because f(x,y) is not defined for floating point y. How could you take an argmax along a fraActional axis? The gradient with respect to x is 0, because f(x+epsilon, y) = f(x) almost everywhere.
4. **f(x,y) = a vector with y elements, each of which taking on the value x** The grad method should return DisconnectedType()() for y, because the elements of f don't depend on y. Only the shape of f depends on y. You probably also want to implement a connection_pattern method to encode this.
5. **f(x) = int(x) converts float x into an int. g(y) = float(y) converts an integer y into a float.** If the final cost C = 0.5 * g(y) = 0.5 g(f(x)), then the gradient with respect to y will be 0.5, even if y is an integer. However, the gradient with respect to x will be 0, because the output of f is integer-valued.

`connection_pattern(node):`

Sometimes needed for proper operation of gradient.grad().

Returns a list of list of bools.

Op.connection_pattern[input_idx][output_idx] is true if the elements of inputs[input_idx] have an effect on the elements of outputs[output_idx].

The `node` parameter is needed to determine the number of inputs. Some ops such as Subtensor take a variable number of inputs.

If no connection_pattern is specified, gradient.grad will assume that all inputs have some elements connected to some elements of all outputs.

This method conveys two pieces of information that are otherwise not part of the theano graph:

1. Which of the op's inputs are truly ancestors of each of the op's outputs. Suppose an op has two inputs, x and y, and outputs f(x) and g(y). y is not really an ancestor of f, but it appears to be so in the theano graph.
2. Whether the actual elements of each input/output are relevant to a computation. For example, the shape op does not read its input's elements, only its shape metadata. d shape(x) / dx should thus raise a disconnected input exception (if these exceptions are enabled). As another example, the elements of the Alloc op's outputs are not affected by the shape arguments to the Alloc op.

Failing to implement this function for an op that needs it can result in two types of incorrect behavior:

1. gradient.grad erroneously raising a TypeError reporting that a gradient is undefined.
2. gradient.grad failing to raise a ValueError reporting that an input is disconnected.

Even if connection_pattern is not implemented correctly, if gradient.grad returns an expression, that expression will be numerically correct.

R_op(*inputs*, *eval_points*)

Optional, to work with gradient.R_op().

This function implements the application of the R-operator on the function represented by your op. Let assume that function is f, with input x, applying the R-operator means computing the Jacobian of f and right-multiplying it by v, the evaluation point, namely: $\frac{\partial f}{\partial x} v$.

`inputs` are the symbolic variables corresponding to the value of the input where you want to evaluate the jacobian, and `eval_points` are the symbolic variables corresponding to the value you want to right multiply the jacobian with.

Same conventions as for the grad method hold. If your op is not differentiable, you can return None. Note that in contrast to the method `grad()`, for `R_op()` you need to return the same number of outputs as there are ouputs of the op. You can think of it in the following terms. You have all your inputs concatenated into a single vector x. You do the same with the evaluation points (which are as many as inputs and of the shame shape) and obtain another vector v. For each output, you reshape it into a vector, compute the jacobian of that vector with respect to x and multiply it by v. As a last step you reshape each of these vectors you obtained for each outputs (that have the same shape as the outputs) back to their corresponding shapes and return them as the output of the `R_op()` method.

Defining an Op: `mul`

We'll define multiplication as a *binary* operation, even though a multiplication Op could take an arbitrary number of arguments.

First, we'll instantiate a `mul` Op:

```
from theano import gof
mul = gof.Op()
```

make_node

This function must take as many arguments as the operation we are defining is supposed to take as inputs—in this example that would be two. This function ensures that both inputs have the `double` type. Since multiplying two doubles yields a double, this function makes an Apply node with an output Variable of type `double`.

```
def make_node(x, y):
    if x.type != double or y.type != double:
        raise TypeError('mul only works on doubles')
    return gof.Apply(mul, [x, y], [double()])
mul.make_node = make_node
```

The first two lines make sure that both inputs are Variables of the `double` type that we created in the previous section. We would not want to multiply two arbitrary types, it would not make much sense (and we'd be screwed when we implement this in C!)

The last line is the meat of the definition. There we create an Apply node representing the application of Op `mul` to inputs `x` and `y`, giving a Variable instance of type `double` as the output.

Note: Theano relies on the fact that if you call the `make_node` method of Apply's first argument on the inputs passed as the Apply's second argument, the call will not fail and the returned Apply instance will be equivalent. This is how graphs are copied.

---

perform

This code actually computes the function. In our example, the data in `inputs` will be instances of Python's built-in type `float` because this is the type that `double.filter()` will always return, per our own definition. `output_storage` will contain a single storage cell for the multiplication's variable.

```
def perform(node, inputs, output_storage):
    x, y = inputs[0], inputs[1]
    z = output_storage[0]
    z[0] = x * y
mul.perform = perform
```

Here, `z` is a list of one element. By default, `z == [None]`.

Note: It is possible that `z` does not contain `None`. If it contains anything else, Theano guarantees that whatever it contains is what `perform` put there the last time it was called with this particular storage. Furthermore, Theano gives you permission to do whatever you want with `z`'s contents, chiefly reusing it or the memory allocated for it. More information can be found in the *Op* documentation.

Warning: We gave `z` the Theano type `double` in `make_node`, which means that a Python `float` must be put there. You should not put, say, an `int` in `z[0]` because Theano assumes Ops handle typing properly.

Trying out our new Op

In the following code, we use our new Op:

```
>>> x, y = double('x'), double('y')
>>> z = mul(x, y)
>>> f = theano.function([x, y], z)
>>> f(5, 6)
30.0
>>> f(5.6, 6.7)
37.519999999999996
```

Note that there is an implicit call to `double.filter()` on each argument, so if we give integers as inputs they are magically cast to the right type. Now, what if we try this?

```
>>> x = double('x')
>>> z = mul(x, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/u/breuleuo/hg/theano/theano/gof/op.py", line 207, in __call__
  File "<stdin>", line 2, in make_node
AttributeError: 'int' object has no attribute 'type'
```

Automatic Constant Wrapping Well, OK. We'd like our Op to be a bit more flexible. This can be done by modifying `make_node` to accept Python `int` or `float` as `x` and/or `y`:

```
def make_node(x, y):
    if isinstance(x, (int, float)):
        x = gof.Constant(double, x)
    if isinstance(y, (int, float)):
        y = gof.Constant(double, y)
    if x.type != double or y.type != double:
        raise TypeError('mul only works on doubles')
    return gof.Apply(mul, [x, y], [double()])
mul.make_node = make_node
```

Whenever we pass a Python int or float instead of a Variable as `x` or `y`, `make_node` will convert it to *Constant* for us. `gof.Constant` is a *Variable* we statically know the value of.

```
>>> x = double('x')
>>> z = mul(x, 2)
>>> f = theano.function([x], z)
>>> f(10)
20.0
>>> f(3.4)
6.7999999999999998
```

Now the code works the way we want it to.

Note: Most Theano Ops follow this convention of up-casting literal make_node arguments to Constants. This makes typing expressions more natural. If you do not want a constant somewhere in your graph, you have to pass a Variable (like `double('x')` here).

Final version

The above example is pedagogical. When you define other basic arithmetic operations `add`, `sub` and `div`, code for `make_node` can be shared between these Ops. Here is revised implementation of these four arithmetic operators:

```
from theano import gof

class BinaryDoubleOp(gof.Op):

    __props__ = ("name", "fn")

    def __init__(self, name, fn):
        self.name = name
        self.fn = fn

    def make_node(self, x, y):
        if isinstance(x, (int, float)):
            x = gof.Constant(double, x)
        if isinstance(y, (int, float)):
            y = gof.Constant(double, y)
        if x.type != double or y.type != double:
            raise TypeError('%s only works on doubles' % self.name)
        return gof.Apply(self, [x, y], [double()])

    def perform(self, node, inp, out):
        x, y = inp
        z, = out
        z[0] = self.fn(x, y)

    def __str__(self):
        return self.name

add = BinaryDoubleOp(name='add',
                     fn=lambda x, y: x + y)

sub = BinaryDoubleOp(name='sub',
                     fn=lambda x, y: x - y)

mul = BinaryDoubleOp(name='mul',
                     fn=lambda x, y: x * y)

div = BinaryDoubleOp(name='div',
                     fn=lambda x, y: x / y)
```

Instead of working directly on an instance of Op, we create a subclass of Op that we can parametrize. All the operations we define are binary. They all work on two inputs with type `double`. They all return a single Variable of type `double`. Therefore, `make_node` does the same thing for all these operations, except for the Op reference `self` passed as first argument to Apply. We define `perform` using the function `fn` passed in the constructor.

This design is a flexible way to define basic operations without duplicating code. The same way a Type subclass represents a set of structurally similar types (see previous section), an Op subclass represents a set of structurally similar operations: operations that have the same input/output types, operations that only differ in one small detail, etc. If you see common patterns in several Ops that you want to define, it can be a good idea to abstract out what you can. Remember that an Op is just an object which satisfies the contract described above on this page and that you should use all the tools at your disposal to create these objects as efficiently as possible.

Exercise: Make a generic DoubleOp, where the number of arguments can also be given as a parameter.

Views and inplace operations

Theano allows the definition of Ops which return a *view* on one of their inputs or operate *inplace* on one or several inputs. This allows more efficient operations on numpy's `ndarray` data type than would be possible otherwise. However, in order to work correctly, these Ops need to implement an additional interface.

Theano recognizes views and inplace operations specially. It ensures that they are used in a consistent manner and it ensures that operations will be carried in a compatible order.

An unfortunate fact is that it is impossible to return a view on an input with the `double` type or to operate inplace on it (Python floats are immutable). Therefore, we can't make examples of these concepts out of what we've just built. Nonetheless, we will present the concepts:

Views

A "view" on an object `x` is an object `y` which shares memory with `x` in some way. In other words, changing `x` might also change `y` and vice versa. For example, imagine a `vector` structure which contains two fields: an integer length and a pointer to a memory buffer. Suppose we have:

```
x = vector {length: 256,
            address: 0xDEADBEEF}

y = vector {length: 224,
            address: 0xDEADBEEF + 0x10}

z = vector {length: 256,
            address: 0xCAFEBABE}
```

So `x` uses the memory range `0xDEADBEEF - 0xDEADBFEF`, `y` the range `0xDEADBEFF - 0xDEADBFDF` and z the range `0xCAFEBABE - 0xCAFEBBBE`. Since the ranges for `x` and `y` overlap, `y` is considered to be a view of `x` and vice versa.

Suppose you had an Op which took `x` as input and returned `y`. You would need to tell Theano that `y` is a view of `x`. For this purpose, you would set the `view_map` field as follows:

```
myop.view_map = {0: [0]}
```

What this means is that the first output (position 0) is a view of the first input (position 0). Even though the interface allows a list of inputs that are viewed by a given output, this feature is currently unsupported. Here are more examples:

```
myop.view_map = {0: [0]} # first output is a view of first input
myop.view_map = {0: [1]} # first output is a view of second input
myop.view_map = {1: [0]} # second output is a view of first input

myop.view_map = {0: [0], # first output is a view of first input
                 1: [1]} # *AND* second output is a view of second input

myop.view_map = {0: [0], # first output is a view of first input
                 1: [0]} # *AND* second output is *ALSO* a view of first input

myop.view_map = {0: [0, 1]} # THIS IS NOT SUPPORTED YET! Only put a single input number in
```

Inplace operations

An inplace operation is one that modifies one or more of its inputs. For example, the expression `x += y` where `x` and `y` are `numpy.ndarray` instances would normally represent an inplace operation on `x`.

Note: Inplace operations in Theano still work in a functional setting: they need to return the modified input. Symbolically, Theano requires one Variable standing for the input *before* being modified and *another* Variable representing the input *after* being modified. Therefore, code using inplace operations would look like this:

```
x, y = dscalars('x', 'y')
r1 = log(x)

# r2 is x AFTER the add_inplace - x still represents the value before adding y
r2 = add_inplace(x, y)

# r3 is log(x) using the x from BEFORE the add_inplace
# r3 is the SAME as r1, even if we wrote this line after the add_inplace line
# Theano is actually going to compute r3 BEFORE r2
r3 = log(x)

# this is log(x) using the x from AFTER the add_inplace (so it's like log(x + y))
r4 = log(r2)
```

Needless to say, this goes for user-defined inplace operations as well: the modified input must figure in the list of outputs you give to `Apply` in the definition of `make_node`.

Also, for technical reasons but also because they are slightly confusing to use as evidenced by the previous code, Theano does not allow the end user to use inplace operations by default. However, it does allow *optimizations* to substitute them in in a later phase. Therefore, typically, if you define an inplace operation, you will define a pure equivalent and an optimization which subsitutes one for the other. Theano will automatically verify if it is possible to do so and will refuse the substitution if it introduces inconsistencies.

Take the previous definitions of `x`, `y` and `z` and suppose an Op which adds one to every byte of its input. If we give `x` as an input to that Op, it can either allocate a new buffer of the same size as `x` (that could be `z`) and set that new buffer's bytes to the variable of the addition. That would be a normal, *pure* Op. Alternatively, it could add one to each byte *in* the buffer `x`, therefore changing it. That would be an inplace Op.

Theano needs to be notified of this fact. The syntax is similar to that of `view_map`:

```
myop.destroy_map = {0: [0]}
```

What this means is that the first output (position 0) operates inplace on the first input (position 0).

```
myop.destroy_map = {0: [0]} # first output operates inplace on first input
myop.destroy_map = {0: [1]} # first output operates inplace on second input
myop.destroy_map = {1: [0]} # second output operates inplace on first input

myop.destroy_map = {0: [0], # first output operates inplace on first input
                    1: [1]} # *AND* second output operates inplace on second input

myop.destroy_map = {0: [0], # first output operates inplace on first input
                    1: [0]} # *AND* second output *ALSO* operates inplace on first input

myop.destroy_map = {0: [0, 1]} # first output operates inplace on both the first and secon
# unlike for views, the previous line is legal and supported
```

Destructive Operations

While some operations will operate inplace on their inputs, some might simply destroy or corrupt them. For example, an Op could do temporary calculations right in its inputs. If that is the case, Theano also needs to be notified. The way to notify Theano is to assume that some output operated inplace on whatever inputs are changed or corrupted by the Op (even if the output does not technically reuse any of the input(s)'s memory). From there, go to the previous section.

Warning: Failure to correctly mark down views and inplace operations using `view_map` and `destroy_map` can lead to nasty bugs. In the absence of this information, Theano might assume that it is safe to execute an inplace operation on some inputs *before* doing other calculations on the *previous* values of the inputs. For example, in the code: `y = log(x); x2 = add_inplace(x, z)` it is imperative to do the logarithm before the addition (because after the addition, the original x that we wanted to take the logarithm of is gone). If Theano does not know that `add_inplace` changes the value of `x` it might invert the order and that will certainly lead to erroneous computations.
You can often identify an incorrect `view_map` or `destroy_map` by using *debugmode*. *Be sure to use DebugMode when developing a new Op that uses ''view_map'' and/or ''destroy_map''.*

Inplace optimization and DebugMode

It is recommended that during the graph construction, all Ops are not inplace. Then an optimization replaces them with inplace ones. Currently DebugMode checks all optimizations that were tried even if they got rejected. One reason an inplace optimization can get rejected is when there is another Op that is already being applied inplace on the same input. Another reason to reject an inplace optimization is if it would introduce a cycle into the graph.

The problem with DebugMode is that it will trigger a useless error when checking a rejected inplace optimization, since it will lead to wrong results. In order to be able to use DebugMode in more situations, your inplace optimization can pre-check whether it will get rejected by using the

`theano.gof.destroyhandler.fast_inplace_check()` function, that will tell which Ops can be performed inplace. You may then skip the optimization if it is incompatible with this check. Note however that this check does not cover all cases where an optimization may be rejected (it will not detect cycles).

Implementing some specific Ops

This page is a guide on the implementation of some specific types of Ops, and points to some examples of such implementations.

For the random number generating Ops, it explains different possible implementation strategies.

Scalar/Elemwise/Reduction Ops

Implementing a Theano scalar Op allows that scalar operation to be reused by our elemwise operations on tensors. If the scalar operation has C code, the elemwise implementation will automatically have C code too. This will enable the fusion of elemwise operations using your new scalar operation. It can also reuse the GPU elemwise code. It is similar for reduction operations.

For examples of how to add new scalar operations, you can have a look at those 2 pull requests, that add GammaLn and Psi and Gamma scalar Ops.

Be careful about some possible problems in the definition of the `grad` method, and about dependencies that may not be available. In particular, see the following fixes: Fix to grad() methods and impl() methods related to SciPy.

SciPy Ops

We can wrap SciPy functions in Theano. But SciPy is an optional dependency. Here is some code that allows the Op to be optional:

```
try:
    import scipy.linalg
    imported_scipy = True
except ImportError:
    # some ops (e.g. Cholesky, Solve, A_Xinv_b) won't work
    imported_scipy = False

class SomeOp(Op):
    ...
    def make_node(self, x):
        assert imported_scipy, (
        "SciPy not available. SciPy is needed for the SomeOp op.")
        ...

from nose.plugins.skip import SkipTest
class test_SomeOp(utt.InferShapeTester):
    ...
    def test_infer_shape(self):
```

```
        if not imported_scipy:
            raise SkipTest("SciPy needed for the SomeOp op.")
        ...
```

Sparse Ops

There are a few differences to keep in mind if you want to make an op that uses *sparse* inputs or outputs, rather than the usual dense tensors. In particular, in the `make_node()` function, you have to call `theano.sparse.as_sparse_variable(x)` on sparse input variables, instead of `as_tensor_variable(x)`.

Another difference is that you need to use `SparseVariable` and `SparseType` instead of `TensorVariable` and `TensorType`.

Do not forget that we support only sparse matrices (so only 2 dimensions) and (like in SciPy) they do not support broadcasting operations by default (although a few Ops do it when called manually). Also, we support only two formats for sparse type: `csr` and `csc`. So in `make_mode()`, you can create output variables like this:

```
out_format = inputs[0].format  # or 'csr' or 'csc' if the output format is fixed
SparseType(dtype=inputs[0].dtype, format=out_format).make_variable()
```

See the sparse `theano.sparse.basic.Cast` op code for a good example of a sparse op with Python code.

Note: From the definition of CSR and CSC formats, CSR column indices are not necessarily sorted. Likewise for CSC row indices. Use `EnsureSortedIndices` if your code does not support it.

Also, there can be explicit zeros in your inputs. Use `Remove0` or `remove0` to make sure they aren't present in your input if you don't support that.

To remove explicit zeros and make sure indices are sorted, use `clean`.

Sparse Gradient There are 2 types of *gradients* for sparse operations: `normal` gradient and `structured` gradient. Please document what your op implements in its docstring. It is important that the user knows it, and it is not always easy to infer from the code. Also make clear which inputs/outputs are sparse and which ones are dense.

Sparse C code Theano does not have a native C code interface for sparse matrices. The reason is simple: we use the SciPy sparse matrix objects and they don't have a C object. So we use a simple trick: a sparse matrix is made of 4 fields that are NumPy vector arrays: `data`, `indices`, `indptr` and `shape`. So to make an op with C code that has sparse variables as inputs, we actually make an op that takes as input the needed fields of those sparse variables.

You can extract the 4 fields with `theano.sparse.basic.csm_properties()`. You can use `theano.sparse.basic.csm_data()`, `theano.sparse.basic.csm_indices()`, `theano.sparse.basic.csm_indptr()` and `theano.sparse.basic.csm_shape()` to extract the individual fields.

You can look at the AddSD sparse op for an example with C code. It implements the addition of a sparse matrix with a dense matrix.

Sparse Tests You can reuse the test system for tensor variables. To generate the needed sparse variable and data, you can use `theano.sparse.tests.test_basic.sparse_random_inputs()`. It takes many parameters, including parameters for the format (csr or csc), the shape, the dtype, whether to have explicit 0 and whether to have unsorted indices.

Random distribution

We have 3 base random number generators. One that wraps NumPy's random generator, one that implements MRG31k3p and one that wraps CURAND.

The fastest, but less developed, is CURAND. It works only on CUDA-enabled GPUs. It does not work on the CPU and it has fewer random distributions implemented.

The recommended and 2nd faster is MRG. It works on the GPU and CPU and has more implemented distributions.

The slowest is our wrapper on NumPy's random generator.

We explain and provide advice on 3 possibles implementations of new distributions here:

1. Extend our wrapper around NumPy random functions. See this PR as an example.
2. Extend MRG implementation by reusing existing Theano Op. Look into the `theano/sandbox/rng_mrg.py` file and grep for all code about binomial(). This distribution uses the output of the uniform distribution and converts it to a binomial distribution with existing Theano operations. The tests go in `theano/sandbox/test_rng_mrg.py`
3. Extend MRG implementation with a new Op that takes a uniform sample as input. Look in the `theano/sandbox/{rng_mrg,multinomial}.py` file and its test in `theano/sandbox/test_multinomal.py`. This is recommended when current Theano ops aren't well suited to modify the uniform to the target distribution. This can happen in particular if there is a loop or complicated condition.

Note: In all cases, you must reuse the same interface as NumPy for compatibility.

OpenMP Ops

To allow consistent interface of Ops that support OpenMP, we have some helper code. Doing this also allows to enable/disable OpenMP globally or per op for fine-grained control.

Your Op needs to inherit from `theano.gof.OpenMPOp`. If it overrides the `__init__()` method, it must have an `openmp=None` parameter and must call `super(MyOpClass, self).__init__(openmp=openmp)`.

The `OpenMPOp` class also implements `c_compile_args` and `make_thunk`. This makes it add the correct g++ flags to compile with OpenMP. It also disables OpenMP and prints a warning if the version of g++ does not support it.

The Theano flag `openmp` is currently False by default as we do not have code that gets sped up with it. The only current implementation is ConvOp. It speeds up some cases, but slows down others. That is why we disable it by default. But we have all the code to have it enabled by default if there is more than 1 core and the environment variable OMP_NUM_THREADS is not 1. This allows Theano to respect the current convention.

Numba Ops

Want C speed without writing C code for your new Op? You can use Numba to generate the C code for you! Here is an example Op doing that.

Alternate Theano Types

Most ops in Theano are used to manipulate tensors. However, Theano also supports many other variable types. The supported types are listed below, along with pointers to the relevant documentation.

- `TensorType` : Theano type that represents a multidimensional array containing elements that all have the same type. Variables of this Theano type are represented in C as objects of class PyArrayObject.
- *TypedList* : Theano type that represents a typed list (a list where every element in the list has the same Theano type). Variables of this Theano type are represented in C as objects of class PyListObject.
- *Scalar* : Theano type that represents a C primitive type. The C type associated with this Theano type is the represented C primitive itself.
- *SparseType* : Theano type used to represent sparse tensors. There is no equivalent C type for this Theano Type but you can split a sparse variable into its parts as TensorVariables. Those can then be used as inputs to an op with C code.
- `Generic` : Theano type that represents a simple Python Object. Variables of this Theano type are represented in C as objects of class PyObject.
- `CDataType` : Theano type that represents a C data type. The C type associated with this Theano type depends on the data being represented.

Implementing double in C

The previous two sections described how to define a double *Type* and arithmetic operations on that Type, but all of them were implemented in pure Python. In this section we will see how to define the double type in such a way that it can be used by operations implemented in C (which we will define in the section after that).

How does it work?

In order to be C-compatible, a Type must provide a C interface to the Python data that satisfy the constraints it puts forward. In other words, it must define C code that can convert a Python reference into some type suitable for manipulation in C and it must define C code that can convert some C structure in which the C implementation of an operation stores its variables into a reference to an object that can be used from Python and is a valid value for the Type.

For example, in the current example, we have a Type which represents a Python float. First, we will choose a corresponding C type. The natural choice would be the primitive `double` type. Then, we need to write code that will take a `PyObject*`, check that it is a Python `float` and extract its value as a `double`. Finally, we need to write code that will take a C `double` and will build a `PyObject*` of Python type `float` that we can work with from Python. We will be using CPython and thus special care must be given to making sure reference counts are updated properly!

The C code we will write makes use of CPython's C API which you can find here.

What needs to be defined

In order to be C-compatible, a Type must define several additional methods, which all start with the `c_` prefix. The complete list can be found in the documentation for `gof.type.Type`. Here, we'll focus on the most important ones:

class `CLinkerType`

`c_declare`(*name*, *sub*, *check_input=True*)
This must return C code which declares variables. These variables will be available to operations defined in C. You may also write typedefs.

`c_init`(*name*, *sub*)
This must return C code which initializes the variables declared in `c_declare`. Either this or `c_extract` will be called.

`c_extract`(*name*, *sub*, *check_input=True*)
This must return C code which takes a reference to a Python object and initializes the variables declared in `c_declare` to match the Python object's data. Either this or `c_init` will be called.

`c_sync`(*name*, *sub*)
When the computations are done, transfer the variables from the C structure we put them in to the destination Python object. This will only be called for the outputs.

`c_cleanup`(*name*, *sub*)
When we are done using the data, clean up whatever we allocated and decrease the appropriate reference counts.

`c_headers`()
`c_libraries`()
`c_header_dirs`()

c_lib_dirs()
Allows you to specify headers, libraries and associated directories.

c_compile_args()
c_no_compile_args()
Allows to specify special compiler arguments to add/exclude.

c_init_code()
Allows you to specify code that will be executed once when the module is initialized, before anything else is executed. For instance, if a type depends on NumPy's C API, then `'import_array();'` has to be among the snippets returned by `c_init_code()`.

c_support_code()
Allows to add helper functions/structs that the *Type* needs.

c_compiler()
Allows to specify a special compiler. This will force this compiler for the current compilation block (a particular op or the full graph). This is used for the GPU code.

c_code_cache_version()
Should return a tuple of hashable objects like integers. This specifies the version of the code. It is used to cache the compiled code. You MUST change the returned tuple for each change in the code. If you don't want to cache the compiled code return an empty tuple or don't implement it.

Each of these functions take two arguments, `name` and `sub` which must be used to parameterize the C code they return. `name` is a string which is chosen by the compiler to represent a *Variable* of the Type in such a way that there are no name conflicts between different pieces of data. Therefore, all variables declared in `c_declare` should have a name which includes `name`. Furthermore, the name of the variable containing a pointer to the Python object associated to the Variable is `py_<name>`.

`sub`, on the other hand, is a dictionary containing bits of C code suitable for use in certain situations. For instance, `sub['fail']` contains code that should be inserted wherever an error is identified.

`c_declare` and `c_extract` also accept a third `check_input` optional argument. If you want your type to validate its inputs, it must only do it when `check_input` is True.

The example code below should help you understand how everything plays out:

Warning: If some error condition occurs and you want to fail and/or raise an Exception, you must use the `fail` code contained in `sub['fail']` (there is an example in the definition of `c_extract` below). You must *NOT* use the `return` statement anywhere, ever, nor `break` outside of your own loops or `goto` to strange places or anything like that. Failure to comply with this restriction could lead to erratic behavior, segfaults and/or memory leaks because Theano defines its own cleanup system and assumes that you are not meddling with it. Furthermore, advanced operations or types might do code transformations on your code such as inserting it in a loop – in that case they can call your code-generating methods with custom failure code that takes into account what they are doing!

Defining the methods

c_declare

```
def c_declare(name, sub):
    return """
    double %(name)s;
    """ % dict(name = name)
double.c_declare = c_declare
```

Very straightforward. All we need to do is write C code to declare a double. That double will be named whatever is passed to our function in the `name` argument. That will usually be some mangled name like "V0", "V2" or "V92" depending on how many nodes there are in the computation graph and what rank the current node has. This function will be called for all Variables whose type is `double`.

You can declare as many variables as you want there and you can also do typedefs. Make sure that the name of each variable contains the `name` argument in order to avoid name collisions (collisions *will* happen if you don't parameterize the variable names as indicated here). Also note that you cannot declare a variable called `py_<name>` or `storage_<name>` because Theano already defines them.

What you declare there is basically the C interface you are giving to your Type. If you wish people to develop operations that make use of it, it's best to publish it somewhere.

c_init

```
def c_init(name, sub):
    return """
    %(name)s = 0.0;
    """ % dict(name = name)
double.c_init = c_init
```

This function has to initialize the double we declared previously to a suitable value. This is useful if we want to avoid dealing with garbage values, especially if our data type is a pointer. This is not going to be called for all Variables with the `double` type. Indeed, if a Variable is an input that we pass from Python, we will want to extract that input from a Python object, therefore it is the `c_extract` method that will be called instead of `c_init`. You can therefore not assume, when writing `c_extract`, that the initialization has been done (in fact you can assume that it *hasn't* been done).

`c_init` will typically be called on output Variables, but in general you should only assume that either `c_init` or `c_extract` has been called, without knowing for sure which of the two.

c_extract

```
def c_extract(name, sub):
    return """
    if (!PyFloat_Check(py_%(name)s)) {
        PyErr_SetString(PyExc_TypeError, "expected a float");
        %(fail)s
    }
    %(name)s = PyFloat_AsDouble(py_%(name)s);
    """ % dict(name = name, fail = sub['fail'])
double.c_extract = c_extract
```

This method is slightly more sophisticated. What happens here is that we have a reference to a Python object which Theano has placed in `py_%(name)s` where `%(name)s` must be substituted for the name given in the inputs. This special variable is declared by Theano as `PyObject* py_%(name)s` where `PyObject*` is a pointer to a Python object as defined by CPython's C API. This is the reference that

corresponds, on the Python side of things, to a Variable with the `double` type. It is what the end user will give and what he or she expects to get back.

In this example, the user will give a Python `float`. The first thing we should do is verify that what we got is indeed a Python `float`. The `PyFloat_Check` function is provided by CPython's C API and does this for us. If the check fails, we set an exception and then we insert code for failure. The code for failure is in `sub["fail"]` and it basically does a `goto` to cleanup code.

If the check passes then we convert the Python float into a double using the `PyFloat_AsDouble` function (yet again provided by CPython's C API) and we put it in our double variable that we declared previously.

c_sync

```
def c_sync(name, sub):
    return """
    Py_XDECREF(py_%(name)s);
    py_%(name)s = PyFloat_FromDouble(%(name)s);
    if (!py_%(name)s) {
        printf("PyFloat_FromDouble failed on: %%f\\n", %(name)s);
        Py_XINCREF(Py_None);
        py_%(name)s = Py_None;
    }
    """ % dict(name = name)
double.c_sync = c_sync
```

This function is probably the trickiest. What happens here is that we have computed some operation on doubles and we have put the variable into the double variable `%(name)s`. Now, we need to put this data into a Python object that we can manipulate on the Python side of things. This Python object must be put into the `py_%(name)s` variable which Theano recognizes (this is the same pointer we get in c_extract).

Now, that pointer is already a pointer to a valid Python object (unless you or a careless implementer did terribly wrong things with it). If we want to point to another object, we need to tell Python that we don't need the old one anymore, meaning that we need to *decrease the previous object's reference count*. The first line, `Py_XDECREF(py_%(name)s)` does exactly this. If it is forgotten, Python will not be able to reclaim the data even if it is not used anymore and there will be memory leaks! This is especially important if the data you work on is large.

Now that we have decreased the reference count, we call `PyFloat_FromDouble` on our double variable in order to convert it to a Python `float`. This returns a new reference which we assign to `py_%(name)s`. From there Theano will do the rest and the end user will happily see a Python `float` come out of his computations.

The rest of the code is not absolutely necessary and it is basically "good practice". `PyFloat_FromDouble` can return `NULL` on failure. `NULL` is a pretty bad reference to have and neither Python nor Theano like it. If this happens, we change the `NULL` pointer (which will cause us problems) to a pointer to `None` (which is *not* a `NULL` pointer). Since `None` is an object like the others, we need to increase its reference count before we can set a new pointer to it. This situation is unlikely to ever happen, but if it ever does, better safe than sorry.

Warning: I said this already but it really needs to be emphasized that if you are going to change the `py_%(name)s` pointer to point to a new reference, you *must* decrease the reference count of whatever it was pointing to before you do the change. This is only valid if you change the pointer, if you are not going to change the pointer, do *NOT* decrease its reference count!

c_cleanup

```
def c_cleanup(name, sub):
    return ""
double.c_cleanup = c_cleanup
```

We actually have nothing to do here. We declared a double on the stack so the C language will reclaim it for us when its scope ends. We didn't `malloc()` anything so there's nothing to `free()`. Furthermore, the `py_%(name)s` pointer hasn't changed so we don't need to do anything with it. Therefore, we have nothing to cleanup. Sweet!

There are however two important things to keep in mind:

First, note that `c_sync` and `c_cleanup` might be called in sequence, so they need to play nice together. In particular, let's say that you allocate memory in `c_init` or `c_extract` for some reason. You might want to either embed what you allocated to some Python object in `c_sync` or to free it in `c_cleanup`. If you do the former, you don't want to free the allocated storage so you should set the pointer to it to `NULL` to avoid that `c_cleanup` mistakenly frees it. Another option is to declare a variable in `c_declare` that you set to true in `c_sync` to notify `c_cleanup` that `c_sync` was called.

Second, whenever you use `%(fail)s` in `c_extract` or in the code of an *operation*, you can count on `c_cleanup` being called right after that. Therefore, it's important to make sure that `c_cleanup` doesn't depend on any code placed after a reference to `%(fail)s`. Furthermore, because of the way Theano blocks code together, only the variables declared in `c_declare` will be visible in `c_cleanup`!

What the generated C will look like

`c_init` and `c_extract` will only be called if there is a Python object on which we want to apply computations using C code. Conversely, `c_sync` will only be called if we want to communicate the values we have computed to Python, and `c_cleanup` will only be called when we don't need to process the data with C anymore. In other words, the use of these functions for a given Variable depends on the the relationship between Python and C with respect to that Variable. For instance, imagine you define the following function and call it:

```
from theano import function
from theano.tensor import double

x, y, z = double('x'), double('y'), double('z')
a = add(x, y)
b = mul(a, z)
f = function([x, y, z], b)
f(1.0, 2.0, 3.0)
```

Using the CLinker, the code that will be produced will look roughly like this:

```
// BEGIN defined by Theano
PyObject* py_x = ...;
PyObject* py_y = ...;
PyObject* py_z = ...;
PyObject* py_a = ...; // note: this reference won't actually be used for anything
PyObject* py_b = ...;
// END defined by Theano

{
  double x; //c_declare for x
  x = ...; //c_extract for x
  {
    double y; //c_declare for y
    y = ...; //c_extract for y
    {
      double z; //c_declare for z
      z = ...; //c_extract for z
      {
        double a; //c_declare for a
        a = 0; //c_init for a
        {
          double b; //c_declare for b
          b = 0; //c_init for b
          {
            a = x + y; //c_code for add
            {
              b = a * z; //c_code for mul
            labelmul:
              //c_cleanup for mul
            }
          labeladd:
            //c_cleanup for add
          }
        labelb:
          py_b = ...; //c_sync for b
          //c_cleanup for b
        }
      labela:
        //c_cleanup for a
      }
    labelz:
      //c_cleanup for z
    }
  labely:
    //c_cleanup for y
  }
labelx:
  //c_cleanup for x
}
```

It's not pretty, but it gives you an idea of how things work (note that the variable names won't be `x`, `y`, `z`, etc. - they will get a unique mangled name). The `fail` code runs a `goto` to the appropriate label in order to run all cleanup that needs to be done. Note which variables get extracted (the three inputs `x`, `y` and `z`),

which ones only get initialized (the temporary variable `a` and the output `b`) and which one is synced (the final output `b`).

The C code above is a single C block for the whole graph. Depending on which *linker* is used to process the computation graph, it is possible that one such block is generated for each operation and that we transit through Python after each operation. In that situation, `a` would be synced by the addition block and extracted by the multiplication block.

Final version

```
from theano import gof

class Double(gof.Type):

    def filter(self, x, strict=False, allow_downcast=None):
        if strict and not isinstance(x, float):
            raise TypeError('Expected a float!')
        return float(x)

    def values_eq_approx(self, x, y, tolerance=1e-4):
        return abs(x - y) / (x + y) < tolerance

    def __str__(self):
        return "double"

    def c_declare(self, name, sub):
        return """
        double %(name)s;
        """ % dict(name = name)

    def c_init(self, name, sub):
        return """
        %(name)s = 0.0;
        """ % dict(name = name)

    def c_extract(self, name, sub):
        return """
        if (!PyFloat_Check(py_%(name)s)) {
            PyErr_SetString(PyExc_TypeError, "expected a float");
            %(fail)s
        }
        %(name)s = PyFloat_AsDouble(py_%(name)s);
        """ % dict(sub, name = name)

    def c_sync(self, name, sub):
        return """
        Py_XDECREF(py_%(name)s);
        py_%(name)s = PyFloat_FromDouble(%(name)s);
        if (!py_%(name)s) {
            printf("PyFloat_FromDouble failed on: %%f\\n", %(name)s);
            Py_XINCREF(Py_None);
            py_%(name)s = Py_None;
```

```
        }
        """ % dict(name = name)

    def c_cleanup(self, name, sub):
        return ""

double = Double()
```

DeepCopyOp

We have an internal Op called DeepCopyOp. It is used to make sure we respect the user vs Theano memory region as described in the *tutorial*. Theano has a Python implementation that calls the object's `copy()` or `deepcopy()` method for Theano types for which it does not know how to generate C code.

You can implement c_code for this op. You register it like this:

```
theano.compile.ops.register_deep_copy_op_c_code(YOUR_TYPE_CLASS, THE_C_CODE, version=())
```

In your C code, you should use %(iname)s and %(oname)s to represent the C variable names of the DeepCopyOp input and output respectively. See an example for the type `CudaNdarrayType` (GPU array) in the file *theano/sandbox/cuda/type.py*. The version parameter is what is returned by DeepCopyOp.c_code_cache_version(). By default, it will recompile the c code for each process.

ViewOp

We have an internal Op called ViewOp. It is used for some verification of inplace/view Ops. Its C implementation increments and decrements Python reference counts, and thus only works with Python objects. If your new type represents Python objects, you should tell ViewOp to generate C code when working with this type, as otherwise it will use Python code instead. This is achieved by calling:

```
theano.compile.ops.register_view_op_c_code(YOUR_TYPE_CLASS, THE_C_CODE, version=())
```

In your C code, you should use %(iname)s and %(oname)s to represent the C variable names of the ViewOp input and output respectively. See an example for the type `CudaNdarrayType` (GPU array) in the file *theano/sandbox/cuda/type.py*. The version parameter is what is returned by ViewOp.c_code_cache_version(). By default, it will recompile the c code for each process.

Shape and Shape_i

We have 2 generic Ops, Shape and Shape_i, that return the shape of any Theano Variable that has a shape attribute (Shape_i returns only one of the elements of the shape).

```
theano.compile.ops.register_shape_c_code(YOUR_TYPE_CLASS, THE_C_CODE, version=())
theano.compile.ops.register_shape_i_c_code(YOUR_TYPE_CLASS, THE_C_CODE, CHECK_INPUT, versi
```

The C code works as the ViewOp. Shape_i has the additional `i` parameter that you can use with `%(i)s`.

In your CHECK_INPUT, you must check that the input has enough dimensions to be able to access the i-th one.

Implementing the arithmetic Ops in C

Now that we have set up our `double` type properly to allow C implementations for operations that work on it, all we have to do now is to actually define these operations in C.

How does it work?

Before a C *Op* is executed, the variables related to each of its inputs will be declared and will be filled appropriately, either from an input provided by the end user (using c_extract) or it might simply have been calculated by another operation. For each of the outputs, the variables associated to them will be declared and initialized.

The operation then has to compute what it needs to using the input variables and place the variables in the output variables.

What needs to be defined

There are less methods to define for an Op than for a Type:

class `Op`

`c_code`(*node*, *name*, *input_names*, *output_names*, *sub*)
This must return C code that carries the computation we want to do.

sub is a dictionary of extras parameters to the c_code method. It contains the following values:

`sub['fail']`

A string of code that you should execute (after ensuring that a python exception is set) if your C code needs to raise an exception.

`sub['context']`

(optional) The name of the variable which holds the context for the node. This will only appear if the op has requested a context by having a `get_context()` method that return something other than None.

`c_code_cleanup`(*node*, *name*, *input_names*, *output_names*, *sub*)
This must return C code that cleans up whatever c_code allocated and that we must free.

Default: The default behavior is to do nothing.

`c_headers`()
Returns a list of headers to include in the file. 'Python.h' is included by default so you don't need to specify it. Also all of the header required by the Types involved (inputs and outputs) will also be included.

`c_header_dirs`()
Returns a list of directories to search for headers (arguments to -I).

c_libraries()
Returns a list of library names that your op needs to link to. All ops are automatically linked with 'python' and the libraries their types require. (arguments to -l)

c_lib_dirs()
Returns a list of directory to search for libraries (arguments to -L).

c_compile_args()
Allows to specify additional arbitrary arguments to g++. This is not usually required.

c_no_compile_args()
Returns a list of g++ arguments that are forbidden when compiling this Op.

c_init_code()
Allows you to specify code that will be executed once when the module is initialized, before anything else is executed. This is for code that will be executed once per Op.

c_init_code_apply(*node*, *name*)
Allows you to specify code that will be executed once when the module is initialized, before anything else is executed and is specialized for a particular apply of an *Op*.

c_init_code_struct(*node*, *name*, *sub*)
Allows you to specify code that will be inserted in the struct constructor of the Op. This is for code which should be executed once per thunk (Apply node, more or less).

sub is a dictionary of extras parameters to the c_code_init_code_struct method. It contains the following values:

`sub['fail']`

A string of code that you should execute (after ensuring that a python exception is set) if your C code needs to raise an exception.

`sub['context']`

(optional) The name of the variable which holds the context for the node. This will only appear if the op has requested a context by having a `get_context()` method that return something other than None.

c_support_code()
Allows you to specify helper functions/structs that the *Op* needs. That code will be reused for each apply of this op. It will be inserted at global scope.

c_support_code_apply(*node*, *name*)
Allows you to specify helper functions/structs specialized for a particular apply of an *Op*. Use `c_support_code()` if the code is the same for each apply of an op. It will be inserted at global scope.

c_support_code_struct(*node*, *name*)
Allows you to specify helper functions of variables that will be specific to one particular thunk. These are inserted at struct scope.

Note You cannot specify CUDA kernels in the code returned by this since that isn't supported by CUDA. You should place your kernels in `c_support_code()` or `c_support_code_apply()` and call them from this code.

c_cleanup_code_struct(*node*, *name*)
Allows you to specify code that will be inserted in the struct destructor of the Op. This is for cleaninp up allocations and stuff like this when the thunk is released (when you "free" a compiled function using this op).

infer_shape(*node*, *(i0_shapes*, *i1_shapes*, *...)*)
Allow optimizations to lift the Shape op over this op. An example of why this is good is when we only need the shape of a variable: we will be able to obtain it without computing the variable itself.

Must return a list where each element is a tuple representing the shape of one output.

For example, for the matrix-matrix product `infer_shape` will have as inputs (node, ((x0,x1), (y0,y1))) and should return [(x0, y1)]. Both the inputs and the return value may be Theano variables.

c_code_cache_version()
Must return a tuple of hashable objects like integers. This specifies the version of the code. It is used to cache the compiled code. You MUST change the returned tuple for each change in the code. If you don't want to cache the compiled code return an empty tuple or don't implement it.

c_code_cache_version_apply(*node*)
Overrides `c_code_cache_version()` if defined, but otherwise has the same contract.

python_constant_folding(*node*)
Optional. If present this method will be called before doing constant folding of a node, with that node as a parameter. If it return True, we will not generate c code when doing constant folding of this node. This is useful when the compilation of the c code will be longer then the computation in python (e.g. Elemwise of scalars).

In addition, this allow to lower the number of compiled module and disk access. Particularly useful when the file system load is high or when theano compilation directory is shared by many process (like on a network file server on a cluster).

get_context(*node*)
(optional) If defined, should return the runtime context the op needs. This context will be passed to the C code through the variable named in *sub['context']*. The variable is also available for use in the code returned by `c_init_code_struct()`. If it returns *None* this is considered the same as if the method was not defined.

If this method is defined and does not return *None*, then the Op *must* have a *context_type* property with the Type to use for the context variable.

_f16_ok
(optional) If this attribute is absent or evaluates to *False*, C code will be disabled for the op if any of its inputs or outputs contains float16 data. This is added as a check to make sure we don't compute wrong results since there is no hardware float16 type so special care must be taken to make sure operations are done correctly.

If you don't intend to deal with float16 data you can leave this undefined.

This attribute is internal and may go away at any point during developpment if a better solution is found.

The `name` argument is currently given an invalid value, so steer away from it. As was the case with Type, `sub['fail']` provides failure code that you *must* use if you want to raise an exception, after setting the exception message.

The `node` argument is an *Apply* node representing an application of the current Op on a list of inputs, producing a list of outputs. `input_names` and `output_names` arguments contain as many strings as there are inputs and outputs to the application of the Op and they correspond to the `name` that is passed to the type of each Variable in these lists. For example, if `node.inputs[0].type == double`, then `input_names[0]` is the `name` argument passed to `double.c_declare` etc. when the first input is processed by Theano.

In a nutshell, `input_names` and `output_names` parameterize the names of the inputs your operation needs to use and the outputs it needs to put variables into. But this will be clear with the examples.

Defining the methods

We will be defining C code for the multiplication Op on doubles.

c_code

```
def c_code(node, name, input_names, output_names, sub):
    x_name, y_name = input_names[0], input_names[1]
    output_name = output_names[0]
    return """
    %(output_name)s = %(x_name)s * %(y_name)s;
    """ % locals()
mul.c_code = c_code
```

And that's it. As we enter the scope of the C code we are defining in the method above, many variables are defined for us. Namely, the variables x_name, y_name and output_name are all of the primitive C `double` type and they were declared using the C code returned by `double.c_declare`.

Implementing multiplication is as simple as multiplying the two input doubles and setting the output double to what comes out of it. If you had more than one output, you would just set the variable(s) for each output to what they should be.

Warning: Do *NOT* use C's `return` statement to return the variable(s) of the computations. Set the output variables directly as shown above. Theano will pick them up for you.

c_code_cleanup

There is nothing to cleanup after multiplying two doubles. Typically, you won't need to define this method unless you malloc() some temporary storage (which you would free() here) or create temporary Python objects (which you would Py_XDECREF() here).

Final version

As before, I tried to organize the code in order to minimize repetition. You can check that mul produces the same C code in this version that it produces in the code I gave above.

```
from theano import gof

class BinaryDoubleOp(gof.Op):

    __props__ = ("name", "fn", "ccode")

    def __init__(self, name, fn, ccode):
        self.name = name
        self.fn = fn
        self.ccode = ccode

    def make_node(self, x, y):
        if isinstance(x, (int, float)):
            x = gof.Constant(double, x)
        if isinstance(y, (int, float)):
            y = gof.Constant(double, y)
        if x.type != double or y.type != double:
            raise TypeError('%s only works on doubles' % self.name)
        return gof.Apply(self, [x, y], [double()])

    def perform(self, node, inp, out):
        x, y = inp
        z, = out
        z[0] = self.fn(x, y)

    def __str__(self):
        return self.name

    def c_code(self, node, name, inp, out, sub):
        x, y = inp
        z, = out
        return self.ccode % locals()


add = BinaryDoubleOp(name='add',
                     fn=lambda x, y: x + y,
                     ccode="%(z)s = %(x)s + %(y)s;")

sub = BinaryDoubleOp(name='sub',
                     fn=lambda x, y: x - y,
                     ccode="%(z)s = %(x)s - %(y)s;")

mul = BinaryDoubleOp(name='mul',
                     fn=lambda x, y: x * y,
                     ccode="%(z)s = %(x)s * %(y)s;")

div = BinaryDoubleOp(name='div',
                     fn=lambda x, y: x / y,
                     ccode="%(z)s = %(x)s / %(y)s;")
```

Graph optimization

In this section we will define a couple optimizations on doubles.

Todo

This tutorial goes way too far under the hood, for someone who just wants to add yet another pattern to the libraries in tensor.opt for example.

We need another tutorial that covers the decorator syntax, and explains how to register your optimization right away. That's what you need to get going.

Later, the rest is more useful for when that decorator syntax type thing doesn't work. (There are optimizations that don't fit that model).

Note: The optimization tag *cxx_only* is used for optimizations that insert Ops which have no Python implementation (so they only have C code). Optimizations with this tag are skipped when there is no C++ compiler available.

Global and local optimizations

First, let's lay out the way optimizations work in Theano. There are two types of optimizations: *global* optimizations and *local* optimizations. A global optimization takes a `FunctionGraph` object (a FunctionGraph is a wrapper around a whole computation graph, you can see its `documentation` for more details) and navigates through it in a suitable way, replacing some Variables by others in the process. A local optimization, on the other hand, is defined as a function on a *single* *Apply* node and must return either `False` (to mean that nothing is to be done) or a list of new Variables that we would like to replace the node's outputs with. A *Navigator* is a special kind of global optimization which navigates the computation graph in some fashion (in topological order, reverse-topological order, random order, etc.) and applies one or more local optimizations at each step.

Optimizations which are holistic, meaning that they must take into account dependencies that might be all over the graph, should be global. Optimizations that can be done with a narrow perspective are better defined as local optimizations. The majority of optimizations we want to define are local.

Global optimization A global optimization (or optimizer) is an object which defines the following methods:

class `Optimizer`

`apply` (*fgraph*)
: This method takes a FunctionGraph object which contains the computation graph and does modifications in line with what the optimization is meant to do. This is one of the main methods of the optimizer.

`add_requirements` (*fgraph*)
: This method takes a FunctionGraph object and adds *features* to it. These features are "plugins"

that are needed for the `apply` method to do its job properly.

optimize(*fgraph*)
This is the interface function called by Theano.

Default: this is defined by Optimizer as `add_requirement(fgraph); apply(fgraph).`

See the section about `FunctionGraph` to understand how to define these methods.

Local optimization A local optimization is an object which defines the following methods:

class LocalOptimizer

transform(*node*)
This method takes an *Apply* node and returns either `False` to signify that no changes are to be done or a list of Variables which matches the length of the node's `outputs` list. When the LocalOptimizer is applied by a Navigator, the outputs of the node passed as argument to the LocalOptimizer will be replaced by the list returned.

One simplification rule

For starters, let's define the following simplification:

$$\frac{xy}{y} = x$$

We will implement it in three ways: using a global optimization, a local optimization with a Navigator and then using the PatternSub facility.

Global optimization Here is the code for a global optimization implementing the simplification described above:

```
from theano.gof import toolbox

class Simplify(gof.Optimizer):
    def add_requirements(self, fgraph):
        fgraph.attach_feature(toolbox.ReplaceValidate())
    def apply(self, fgraph):
        for node in fgraph.toposort():
            if node.op == div:
                x, y = node.inputs
                z = node.outputs[0]
                if x.owner and x.owner.op == mul:
                    a, b = x.owner.inputs
                    if y == a:
                        fgraph.replace_validate(z, b)
                    elif y == b:
                        fgraph.replace_validate(z, a)

simplify = Simplify()
```

Todo

What is add_requirements? Why would we know to do this? Are there other requirements we might want to know about?

Here's how it works: first, in `add_requirements`, we add the `ReplaceValidate` *FunctionGraph Features* located in *toolbox – [doc TODO]*. This feature adds the `replace_validate` method to `fgraph`, which is an enhanced version of `replace` that does additional checks to ensure that we are not messing up the computation graph (note: if `ReplaceValidate` was already added by another optimizer, `extend` will do nothing). In a nutshell, `toolbox.ReplaceValidate` grants access to `fgraph.replace_validate`, and `fgraph.replace_validate` allows us to replace a Variable with another while respecting certain validation constraints. You can browse the list of *FunctionGraph Feature List* and see if some of them might be useful to write optimizations with. For example, as an exercise, try to rewrite Simplify using `NodeFinder`. (Hint: you want to use the method it publishes instead of the call to toposort!)

Then, in `apply` we do the actual job of simplification. We start by iterating through the graph in topological order. For each node encountered, we check if it's a `div` node. If not, we have nothing to do here. If so, we put in `x`, `y` and `z` the numerator, denominator and quotient (output) of the division. The simplification only occurs when the numerator is a multiplication, so we check for that. If the numerator is a multiplication we put the two operands in `a` and `b`, so we can now say that `z == (a*b)/y`. If `y==a` then `z==b` and if `y==b` then `z==a`. When either case happens then we can replace `z` by either `a` or `b` using `fgraph.replace_validate` - else we do nothing. You might want to check the documentation about *Variable* and *Apply* to get a better understanding of the pointer-following game you need to get ahold of the nodes of interest for the simplification (`x`, `y`, `z`, `a`, `b`, etc.).

Test time:

```
>>> x = double('x')
>>> y = double('y')
>>> z = double('z')
>>> a = add(z, mul(div(mul(y, x), y), div(z, x)))
>>> e = gof.FunctionGraph([x, y, z], [a])
>>> e
[add(z, mul(div(mul(y, x), y), div(z, x)))]
>>> simplify.optimize(e)
>>> e
[add(z, mul(x, div(z, x)))]
```

Cool! It seems to work. You can check what happens if you put many instances of $\frac{xy}{y}$ in the graph. Note that it sometimes won't work for reasons that have nothing to do with the quality of the optimization you wrote. For example, consider the following:

```
>>> x = double('x')
>>> y = double('y')
>>> z = double('z')
>>> a = div(mul(add(y, z), x), add(y, z))
>>> e = gof.FunctionGraph([x, y, z], [a])
>>> e
[div(mul(add(y, z), x), add(y, z))]
>>> simplify.optimize(e)
```

```
>>> e
[div(mul(add(y, z), x), add(y, z))]
```

Nothing happened here. The reason is: `add(y, z) != add(y, z)`. That is the case for efficiency reasons. To fix this problem we first need to merge the parts of the graph that represent the same computation, using the `merge_optimizer` defined in `theano.gof.opt`.

```
>>> from theano.gof.opt import merge_optimizer
>>> merge_optimizer.optimize(e)
>>> e
[div(mul(*1 -> add(y, z), x), *1)]
>>> simplify.optimize(e)
>>> e
[x]
```

Once the merge is done, both occurrences of `add(y, z)` are collapsed into a single one and is used as an input in two places. Note that `add(x, y)` and `add(y, x)` are still considered to be different because Theano has no clue that `add` is commutative. You may write your own global optimizer to identify computations that are identical with full knowledge of the rules of arithmetics that your Ops implement. Theano might provide facilities for this somewhere in the future.

Note: `FunctionGraph` is a Theano structure intended for the optimization phase. It is used internally by function and is rarely exposed to the end user. You can use it to test out optimizations, etc. if you are comfortable with it, but it is recommended to use the function frontend and to interface optimizations with `optdb` (we'll see how to do that soon).

Local optimization The local version of the above code would be the following:

```
class LocalSimplify(gof.LocalOptimizer):
    def transform(self, node):
        if node.op == div:
            x, y = node.inputs
            if x.owner and x.owner.op == mul:
                a, b = x.owner.inputs
                if y == a:
                    return [b]
                elif y == b:
                    return [a]
        return False
    def tracks(self):
        # This should be needed for the EquilibriumOptimizer
        # but it isn't now
        # TODO: do this and explain it
        return [] # that's not what you should do

local_simplify = LocalSimplify()
```

Todo

Fix up previous example... it's bad and incomplete.

The definition of transform is the inner loop of the global optimizer, where the node is given as argument. If no changes are to be made, `False` must be returned. Else, a list of what to replace the node's outputs with must be returned. This list must have the same length as node.ouputs. If one of node.outputs don't have clients(it is not used in the graph), you can put None in the returned list to remove it.

In order to apply the local optimizer we must use it in conjunction with a *Navigator*. Basically, a *Navigator* is a global optimizer that loops through all nodes in the graph (or a well-defined subset of them) and applies one or several local optimizers on them.

```
>>> x = double('x')
>>> y = double('y')
>>> z = double('z')
>>> a = add(z, mul(div(mul(y, x), y), div(z, x)))
>>> e = gof.FunctionGraph([x, y, z], [a])
>>> e
[add(z, mul(div(mul(y, x), y), div(z, x)))]
>>> simplify = gof.TopoOptimizer(local_simplify)
>>> simplify.optimize(e)
>>> e
[add(z, mul(x, div(z, x)))]
```

OpSub, OpRemove, PatternSub Theano defines some shortcuts to make LocalOptimizers:

OpSub(*op1*, *op2*)
: Replaces all uses of *op1* by *op2*. In other words, the outputs of all *Apply* involving *op1* by the outputs of Apply nodes involving *op2*, where their inputs are the same.

OpRemove(*op*)
: Removes all uses of *op* in the following way: if `y = op(x)` then `y` is replaced by `x`. *op* must have as many outputs as it has inputs. The first output becomes the first input, the second output becomes the second input, and so on.

PatternSub(*pattern1*, *pattern2*)
: Replaces all occurrences of the first pattern by the second pattern. See `PatternSub`.

```
from theano.gof.opt import OpSub, OpRemove, PatternSub

# Replacing add by mul (this is not recommended for primarily
# mathematical reasons):
add_to_mul = OpSub(add, mul)

# Removing identity
remove_identity = OpRemove(identity)

# The "simplify" operation we've been defining in the past few
# sections. Note that we need two patterns to account for the
# permutations of the arguments to mul.
local_simplify_1 = PatternSub((div, (mul, 'x', 'y'), 'y'),
                              'x')
local_simplify_2 = PatternSub((div, (mul, 'x', 'y'), 'x'),
                              'y')
```

Note: `OpSub`, `OpRemove` and `PatternSub` produce local optimizers, which means that everything we said previously about local optimizers apply: they need to be wrapped in a Navigator, etc.

Todo

wtf is a navigator?

When an optimization can be naturally expressed using `OpSub`, `OpRemove` or `PatternSub`, it is highly recommended to use them.

WRITEME: more about using PatternSub (syntax for the patterns, how to use constraints, etc. - there's some decent doc at `PatternSub` for those interested)

The optimization database (optdb)

Theano exports a symbol called `optdb` which acts as a sort of ordered database of optimizations. When you make a new optimization, you must insert it at the proper place in the database. Furthermore, you can give each optimization in the database a set of tags that can serve as a basis for filtering.

The point of optdb is that you might want to apply many optimizations to a computation graph in many unique patterns. For example, you might want to do optimization X, then optimization Y, then optimization Z. And then maybe optimization Y is an EquilibriumOptimizer containing LocalOptimizers A, B and C which are applied on every node of the graph until they all fail to change it. If some optimizations act up, we want an easy way to turn them off. Ditto if some optimizations are very CPU-intensive and we don't want to take the time to apply them.

The optdb system allows us to tag each optimization with a unique name as well as informative tags such as 'stable', 'buggy' or 'cpu_intensive', all this without compromising the structure of the optimizations.

Definition of optdb optdb is an object which is an instance of `SequenceDB`, itself a subclass of `DB`. There exist (for now) two types of DB, SequenceDB and EquilibriumDB. When given an appropriate Query, DB objects build an Optimizer matching the query.

A SequenceDB contains Optimizer or DB objects. Each of them has a name, an arbitrary number of tags and an integer representing their order in the sequence. When a Query is applied to a SequenceDB, all Optimizers whose tags match the query are inserted in proper order in a SequenceOptimizer, which is returned. If the SequenceDB contains DB instances, the Query will be passed to them as well and the optimizers they return will be put in their places.

An EquilibriumDB contains LocalOptimizer or DB objects. Each of them has a name and an arbitrary number of tags. When a Query is applied to an EquilibriumDB, all LocalOptimizers that match the query are inserted into an EquilibriumOptimizer, which is returned. If the SequenceDB contains DB instances, the Query will be passed to them as well and the LocalOptimizers they return will be put in their places (note that as of yet no DB can produce LocalOptimizer objects, so this is a moot point).

Theano contains one principal DB object, `optdb`, which contains all of Theano's optimizers with proper tags. It is recommended to insert new Optimizers in it. As mentioned previously, optdb is a SequenceDB, so, at the top level, Theano applies a sequence of global optimizations to the computation graphs.

Query A Query is built by the following call:

```
theano.gof.Query(include, require = None, exclude = None, subquery = None)
```

class Query

include
A set of tags (a tag being a string) such that every optimization obtained through this Query must have **one** of the tags listed. This field is required and basically acts as a starting point for the search.

require
A set of tags such that every optimization obtained through this Query must have **all** of these tags.

exclude
A set of tags such that every optimization obtained through this Query must have **none** of these tags.

subquery
optdb can contain sub-databases; subquery is a dictionary mapping the name of a sub-database to a special Query. If no subquery is given for a sub-database, the original Query will be used again.

Furthermore, a Query object includes three methods, `including`, `requiring` and `excluding` which each produce a new Query object with include, require and exclude sets refined to contain the new [WRITEME]

Examples Here are a few examples of how to use a Query on optdb to produce an Optimizer:

```
from theano.compile import optdb

# This is how the optimizer for the fast_run mode is defined
fast_run = optdb.query(Query(include = ['fast_run']))

# This is how the optimizer for the fast_compile mode is defined
fast_compile = optdb.query(Query(include = ['fast_compile']))

# This is the same as fast_run but no optimizations will replace
# any operation by an inplace version. This assumes, of course,
# that all inplace operations are tagged as 'inplace' (as they
# should!)
fast_run_no_inplace = optdb.query(Query(include = ['fast_run'], exclude = ['inplace']))
fast_run_no_inplace = fast_run.excluding('inplace')
```

Registering an Optimizer Let's say we have a global optimizer called `simplify`. We can add it to `optdb` as follows:

```
# optdb.register(name, optimizer, order, *tags)
optdb.register('simplify', simplify, 0.5, 'fast_run')
```

Once this is done, the FAST_RUN mode will automatically include your optimization (since you gave it the 'fast_run' tag). Of course, already-compiled functions will see no change. The 'order' parameter (what it means and how to choose it) will be explained in *optdb structure* below.

Registering a LocalOptimizer LocalOptimizers may be registered in two ways:

- Wrap them in a Navigator and insert them like a global optimizer (see previous section).
- Put them in an EquilibriumDB.

Theano defines two EquilibriumDBs where you can put local optimizations:

`canonicalize()`
This contains optimizations that aim to *simplify* the graph:

•Replace rare or esoterical operations with their equivalents using elementary operations.

•Order operations in a canonical way (any sequence of multiplications and divisions can be rewritten to contain at most one division, for example; `x*x` can be rewritten `x**2`; etc.)

•Fold constants (`Constant(2)*Constant(2)` becomes `Constant(4)`)

`specialize()`
This contains optimizations that aim to *specialize* the graph:

•Replace a combination of operations with a special operation that does the same thing (but better).

For each group, all optimizations of the group that are selected by the Query will be applied on the graph over and over again until none of them is applicable, so keep that in mind when designing it: check carefully that your optimization leads to a fixpoint (a point where it cannot apply anymore) at which point it returns `False` to indicate its job is done. Also be careful not to undo the work of another local optimizer in the group, because then the graph will oscillate between two or more states and nothing will get done.

optdb structure optdb contains the following Optimizers and sub-DBs, with the given priorities and tags:

Order	Name	Description
0	merge1	First merge operation
1	canonicalize	Simplify the graph
2	specialize	Add specialized operations
49	merge2	Second merge operation
49.5	add_destroy_handler	Enable inplace optimizations
100	merge3	Third merge operation

The merge operations are meant to put together parts of the graph that represent the same computation. Since optimizations can modify the graph in such a way that two previously different-looking parts of the graph become similar, we merge at the beginning, in the middle and at the very end. Technically, we only really need to do it at the end, but doing it in previous steps reduces the size of the graph and therefore increases the efficiency of the process.

See previous section for more information about the canonicalize and specialize steps.

The `add_destroy_handler` step is not really an optimization. It is a marker. Basically:

Warning: Any optimization which inserts inplace operations in the computation graph must appear **after** the `add_destroy_handler` "optimizer". In other words, the priority of any such optimization must be **>= 50**. Failure to comply by this restriction can lead to the creation of incorrect computation graphs.

The reason the destroy handler is not inserted at the beginning is that it is costly to run. It is cheaper to run most optimizations under the assumption there are no inplace operations.

Navigator WRITEME

Profiling Theano function compilation

You find that compiling a Theano function is taking too much time? You can get profiling information about Theano optimization. The normal *Theano profiler* will provide you with very high-level information. The indentation shows the included in/subset relationship between sections. The top of its output look like this:

```
Function profiling
==================
  Message: PATH_TO_A_FILE:23
  Time in 0 calls to Function.__call__: 0.000000e+00s
  Total compile time: 1.131874e+01s
    Number of Apply nodes: 50
    Theano Optimizer time: 1.152431e+00s
       Theano validate time: 2.790451e-02s
    Theano Linker time (includes C, CUDA code generation/compiling): 7.893991e-02s
       Import time 1.153541e-02s
  Time in all call to theano.grad() 4.732513e-02s
```

Explanations:

- `Total compile time: 1.131874e+01s` gives the total time spent inside *theano.function*.
- `Number of Apply nodes: 50` means that after optimization, there are 50 apply node in the graph.
- `Theano Optimizer time: 1.152431e+00s` means that we spend 1.15s in the `theano.function` phase where we optimize (modify) the graph to make it faster / more stable numerically / work on GPU /...
- `Theano validate time: 2.790451e-02s` means that we spent 2.8e-2s in the *validate* subset of the optimization phase.
- `Theano Linker time (includes C, CUDA code generation/compiling): 7.893991e-02s` means that we spent 7.9e-2s in *linker* phase of `theano.function`.
- `Import time 1.153541e-02s` is a subset of the linker time where we import the compiled module.
- `Time in all call to theano.grad() 4.732513e-02s` tells that we spent a total of 4.7e-2s in all calls to `theano.grad`. This is outside of the calls to `theano.function`.

The *linker* phase includes the generation of the C code, the time spent by g++ to compile and the time needed by Theano to build the object we return. The C code generation and compilation is cached, so the first time you compile a function and the following ones could take different amount of execution time.

Detailed profiling of Theano optimizer You can get more detailed profiling information about the Theano optimizer phase by setting to *True* the Theano flags `config.profile_optimizer`.

This will output something like this:

```
Optimizer Profile
-----------------
 SeqOptimizer  OPT_FAST_RUN  time 1.152s for 123/50 nodes before/after optimization
   0.028s for fgraph.validate()
   0.131s for callback
   time      - (name, class, index) - validate time
   0.751816s - ('canonicalize', 'EquilibriumOptimizer', 4) - 0.004s
     EquilibriumOptimizer      canonicalize
       time 0.751s for 14 passes
       nb nodes (start, end,  max) 108 81 117
       time io_toposort 0.029s
       time in local optimizers 0.687s
       time in global optimizers 0.010s
        0 - 0.050s 27 (0.000s in global opts, 0.002s io_toposort) - 108 nodes - ('local_dir
        1 - 0.288s 26 (0.002s in global opts, 0.002s io_toposort) - 117 nodes - ('local_dir
        2 - 0.044s 13 (0.002s in global opts, 0.003s io_toposort) - 96 nodes - ('constant_
        3 - 0.045s 11 (0.000s in global opts, 0.002s io_toposort) - 91 nodes - ('constant_
        4 - 0.035s 8 (0.002s in global opts, 0.002s io_toposort) - 93 nodes - ('local_fill_
        5 - 0.035s 6 (0.000s in global opts, 0.002s io_toposort) - 88 nodes - ('local_fill_
        6 - 0.038s 10 (0.001s in global opts, 0.002s io_toposort) - 95 nodes - ('local_fill
        7 - 0.032s 5 (0.001s in global opts, 0.002s io_toposort) - 91 nodes - ('local_fill_
        8 - 0.034s 5 (0.000s in global opts, 0.002s io_toposort) - 92 nodes - ('local_fill_
        9 - 0.031s 6 (0.001s in global opts, 0.002s io_toposort) - 90 nodes - ('local_fill_
       10 - 0.032s 5 (0.000s in global opts, 0.002s io_toposort) - 89 nodes - ('local_dimsh
       11 - 0.030s 5 (0.000s in global opts, 0.002s io_toposort) - 88 nodes - ('local_dimsh
       12 - 0.026s 1 (0.000s in global opts, 0.003s io_toposort) - 81 nodes - ('MergeOptimi
       13 - 0.031s 0 (0.000s in global opts, 0.003s io_toposort) - 81 nodes -
       times - times applied - nb node created - name:
       0.263s - 15 - 0 - constant_folding
       0.096s - 2 - 14 - local_greedy_distributor
       0.066s - 4 - 19 - local_mul_canonizer
       0.046s - 28 - 57 - local_fill_sink
       0.042s - 35 - 78 - local_dimshuffle_lift
       0.018s - 5 - 15 - local_upcast_elemwise_constant_inputs
       0.010s - 11 - 4 - MergeOptimizer
       0.009s - 4 - 0 - local_useless_elemwise
       0.005s - 11 - 2 - local_fill_to_alloc
       0.004s - 3 - 6 - local_neg_to_mul
       0.002s - 1 - 3 - local_lift_transpose_through_dot
       0.002s - 3 - 4 - local_shape_to_shape_i
       0.002s - 2 - 4 - local_subtensor_lift
       0.001s - 3 - 0 - local_subtensor_make_vector
       0.001s - 1 - 1 - local_sum_all_to_none
       0.131s - in 62 optimization that where not used (display only those with a runtime >
```

```
      0.050s - local_add_canonizer
      0.018s - local_mul_zero
      0.016s - local_one_minus_erf
      0.010s - local_func_inv
      0.006s - local_0_dot_x
      0.005s - local_track_shape_i
      0.004s - local_mul_switch_sink
      0.004s - local_fill_cut
      0.004s - local_one_minus_erf2
      0.003s - local_remove_switch_const_cond
      0.003s - local_cast_cast
      0.002s - local_IncSubtensor_serialize
      0.001s - local_sum_div_dimshuffle
      0.001s - local_div_switch_sink
      0.001s - local_dimshuffle_no_inplace_at_canonicalize
      0.001s - local_cut_useless_reduce
      0.001s - local_reduce_join
      0.000s - local_sum_sum
      0.000s - local_useless_alloc
      0.000s - local_reshape_chain
      0.000s - local_useless_subtensor
      0.000s - local_reshape_lift
      0.000s - local_flatten_lift
      0.000s - local_useless_slice
      0.000s - local_subtensor_of_alloc
      0.000s - local_subtensor_of_dot
      0.000s - local_subtensor_merge
0.101733s - ('elemwise_fusion', 'SeqOptimizer', 13) - 0.000s
  SeqOptimizer      elemwise_fusion  time 0.102s for 78/50 nodes before/after optimizati
    0.000s for fgraph.validate()
    0.004s for callback
    0.095307s - ('composite_elemwise_fusion', 'FusionOptimizer', 1) - 0.000s
      FusionOptimizer
       nb_iter 3
       nb_replacement 10
       nb_inconsistency_replace 0
       validate_time 0.000249624252319
       callback_time 0.00316381454468
       time_toposort 0.00375390052795
    0.006412s - ('local_add_mul_fusion', 'FusionOptimizer', 0) - 0.000s
      FusionOptimizer
       nb_iter 2
       nb_replacement 3
       nb_inconsistency_replace 0
       validate_time 6.43730163574e-05
       callback_time 0.000783205032349
       time_toposort 0.0035240650177
0.090089s - ('inplace_elemwise_optimizer', 'FromFunctionOptimizer', 30) - 0.019s
0.048993s - ('BlasOpt', 'SeqOptimizer', 8) - 0.000s
  SeqOptimizer      BlasOpt  time 0.049s for 81/80 nodes before/after optimization
    0.000s for fgraph.validate()
    0.003s for callback
    0.035997s - ('gemm_optimizer', 'GemmOptimizer', 1) - 0.000s
```

```
      GemmOptimizer
       nb_iter 2
       nb_replacement 2
       nb_replacement_didn_t_remove 0
       nb_inconsistency_make 0
       nb_inconsistency_replace 0
       time_canonicalize 0.00720071792603
       time_factor_can 9.05990600586e-06
       time_factor_list 0.00128507614136
       time_toposort 0.00311398506165
       validate_time 4.60147857666e-05
       callback_time 0.00174236297607
    0.004569s - ('local_dot_to_dot22', 'TopoOptimizer', 0) - 0.000s
      TopoOptimizer
        nb_node (start, end, changed) (81, 81, 5)
        init io_toposort 0.00139284133911
        loop time 0.00312399864197
        callback_time 0.00172805786133
    0.002283s - ('local_dot22_to_dot22scalar', 'TopoOptimizer', 2) - 0.000s
      TopoOptimizer
        nb_node (start, end, changed) (80, 80, 0)
        init io_toposort 0.00171804428101
        loop time 0.000502109527588
        callback_time 0.0
    0.002257s - ('local_gemm_to_gemv', 'EquilibriumOptimizer', 3) - 0.000s
      EquilibriumOptimizer          local_gemm_to_gemv
        time 0.002s for 1 passes
        nb nodes (start, end,  max) 80 80 80
        time io_toposort 0.001s
        time in local optimizers 0.000s
        time in global optimizers 0.000s
         0 - 0.002s 0 (0.000s in global opts, 0.001s io_toposort) - 80 nodes -
    0.002227s - ('use_c_blas', 'TopoOptimizer', 4) - 0.000s
      TopoOptimizer
        nb_node (start, end, changed) (80, 80, 0)
        init io_toposort 0.0014750957489
        loop time 0.00068998336792
        callback_time 0.0
    0.001632s - ('use_scipy_ger', 'TopoOptimizer', 5) - 0.000s
      TopoOptimizer
        nb_node (start, end, changed) (80, 80, 0)
        init io_toposort 0.00138401985168
        loop time 0.000202178955078
        callback_time 0.0
0.031740s - ('specialize', 'EquilibriumOptimizer', 9) - 0.000s
  EquilibriumOptimizer      specialize
    time 0.031s for 2 passes
    nb nodes (start, end,  max) 80 78 80
    time io_toposort 0.003s
    time in local optimizers 0.022s
    time in global optimizers 0.004s
     0 - 0.017s 6 (0.002s in global opts, 0.001s io_toposort) - 80 nodes - ('constant_fo
     1 - 0.014s 0 (0.002s in global opts, 0.001s io_toposort) - 78 nodes -
```

```
      times - times applied - nb node created - name:
      0.003s - 1 - 1 - local_mul_specialize
      0.002s - 1 - 2 - local_elemwise_alloc
      0.002s - 2 - 0 - constant_folding
      0.001s - 1 - 1 - local_div_to_inv
      0.001s - 1 - 1 - local_mul_to_sqr
      0.016s - in 69 optimization that where not used (display only those with a runtime >
        0.004s - crossentropy_to_crossentropy_with_softmax_with_bias
        0.002s - local_one_minus_erf
        0.002s - Elemwise{sub,no_inplace}(z, Elemwise{mul,no_inplace}(alpha subject to <fu
        0.002s - local_add_specialize
        0.001s - local_func_inv
        0.001s - local_useless_elemwise
        0.001s - local_abs_merge
        0.001s - local_track_shape_i
        0.000s - local_one_minus_erf2
        0.000s - local_sum_mul_by_scalar
        0.000s - local_elemwise_sub_zeros
        0.000s - local_cast_cast
        0.000s - local_alloc_unary
        0.000s - Elemwise{log,no_inplace}(Softmax(x)) -> <function make_out_pattern at 0x7
        0.000s - local_sum_div_dimshuffle
        0.000s - local_sum_alloc
        0.000s - local_dimshuffle_lift
        0.000s - local_reduce_broadcastable
        0.000s - local_grad_log_erfc_neg
        0.000s - local_advanced_indexing_crossentropy_onehot
        0.000s - local_log_erfc
        0.000s - local_log1p
        0.000s - local_log_add
        0.000s - local_useless_alloc
        0.000s - local_neg_neg
        0.000s - local_neg_div_neg
...
```

To understand this profile here is some explanation of how optimizations work:

- Optimizations are organized in an hierarchy. At the top level, there is a `SeqOptimizer` (Sequence Optimizer). It contains other optimizers, and applies them in the order they were specified. Those sub-optimizers can be of other types, but are all *global* optimizers.
- Each Optimizer in the hierarchy will print some stats about itself. The information that it prints depends of the type of the optimizer.
- The SeqOptimizer will print some stats at the start:

```
      Optimizer Profile
      -----------------
       SeqOptimizer  OPT_FAST_RUN  time 1.152s for 123/50 nodes before/after optimizat
         0.028s for fgraph.validate()
         0.131s for callback
         time      - (name, class, index) - validate time
```

Then it will print, with some additional indentation, each sub-optimizer's profile

```
information. These sub-profiles are ordered by the time they took to execute,
not by their execution order.
```

  - `OPT_FAST_RUN` is the name of the optimizer
  - 1.152s is the total time spent in that optimizer
  - 123/50 means that before this optimization, there were 123 apply node in the function graph, and after only 50.
  - 0.028s means it spent that time calls to `fgraph.validate()`
  - 0.131s means it spent that time for callbacks. This is a mechanism that can trigger other execution when there is a change to the FunctionGraph.
  - `time - (name, class, index) - validate time` tells how the information for each sub-optimizer get printed.
  - All other instances of `SeqOptimizer` are described like this. In particular, some sub-optimizer from OPT_FAST_RUN that are also `SeqOptimizer`.

- The `SeqOptimizer` will print some stats at the start:

```
0.751816s - ('canonicalize', 'EquilibriumOptimizer', 4) - 0.004s
  EquilibriumOptimizer      canonicalize
    time 0.751s for 14 passes
    nb nodes (start, end,  max) 108 81 117
    time io_toposort 0.029s
    time in local optimizers 0.687s
    time in global optimizers 0.010s
     0 - 0.050s 27 (0.000s in global opts, 0.002s io_toposort) - 108 nodes - ('loc
     1 - 0.288s 26 (0.002s in global opts, 0.002s io_toposort) - 117 nodes - ('loc
     2 - 0.044s 13 (0.002s in global opts, 0.003s io_toposort) - 96 nodes - ('cons
     3 - 0.045s 11 (0.000s in global opts, 0.002s io_toposort) - 91 nodes - ('cons
     4 - 0.035s 8 (0.002s in global opts, 0.002s io_toposort) - 93 nodes - ('local
     5 - 0.035s 6 (0.000s in global opts, 0.002s io_toposort) - 88 nodes - ('local
     6 - 0.038s 10 (0.001s in global opts, 0.002s io_toposort) - 95 nodes - ('loca
     7 - 0.032s 5 (0.001s in global opts, 0.002s io_toposort) - 91 nodes - ('local
     8 - 0.034s 5 (0.000s in global opts, 0.002s io_toposort) - 92 nodes - ('local
     9 - 0.031s 6 (0.001s in global opts, 0.002s io_toposort) - 90 nodes - ('local
    10 - 0.032s 5 (0.000s in global opts, 0.002s io_toposort) - 89 nodes - ('local
    11 - 0.030s 5 (0.000s in global opts, 0.002s io_toposort) - 88 nodes - ('local
    12 - 0.026s 1 (0.000s in global opts, 0.003s io_toposort) - 81 nodes - ('Merge
    13 - 0.031s 0 (0.000s in global opts, 0.003s io_toposort) - 81 nodes -
    times - times applied - nb node created - name:
    0.263s - 15 - 0 - constant_folding
    0.096s - 2 - 14 - local_greedy_distributor
    0.066s - 4 - 19 - local_mul_canonizer
    0.046s - 28 - 57 - local_fill_sink
    0.042s - 35 - 78 - local_dimshuffle_lift
    0.018s - 5 - 15 - local_upcast_elemwise_constant_inputs
    0.010s - 11 - 4 - MergeOptimizer
    0.009s - 4 - 0 - local_useless_elemwise
    0.005s - 11 - 2 - local_fill_to_alloc
    0.004s - 3 - 6 - local_neg_to_mul
```

```
      0.002s - 1 - 3 - local_lift_transpose_through_dot
      0.002s - 3 - 4 - local_shape_to_shape_i
      0.002s - 2 - 4 - local_subtensor_lift
      0.001s - 3 - 0 - local_subtensor_make_vector
      0.001s - 1 - 1 - local_sum_all_to_none
      0.131s - in 62 optimization that where not used (display only those with a run
        0.050s - local_add_canonizer
        0.018s - local_mul_zero
        0.016s - local_one_minus_erf
        0.010s - local_func_inv
        0.006s - local_0_dot_x
        0.005s - local_track_shape_i
        0.004s - local_mul_switch_sink
        0.004s - local_fill_cut
        0.004s - local_one_minus_erf2
        0.003s - local_remove_switch_const_cond
        0.003s - local_cast_cast
        0.002s - local_IncSubtensor_serialize
        0.001s - local_sum_div_dimshuffle
        0.001s - local_div_switch_sink
        0.001s - local_dimshuffle_no_inplace_at_canonicalize
        0.001s - local_cut_useless_reduce
        0.001s - local_reduce_join
        0.000s - local_sum_sum
        0.000s - local_useless_alloc
        0.000s - local_reshape_chain
        0.000s - local_useless_subtensor
        0.000s - local_reshape_lift
        0.000s - local_flatten_lift
        0.000s - local_useless_slice
        0.000s - local_subtensor_of_alloc
        0.000s - local_subtensor_of_dot
        0.000s - local_subtensor_merge
```

- `0.751816s - ('canonicalize', 'EquilibriumOptimizer', 4) - 0.004s` This line is from `SeqOptimizer`, and indicates information related to a sub-optimizer. It means that this sub-optimizer took a total of .7s. Its name is `'canonicalize'`. It is an `EquilibriumOptimizer`. It was executed at index 4 by the `SeqOptimizer`. It spent 0.004s in the *validate* phase.
- All other lines are from the profiler of the `EquilibriumOptimizer`.
- An `EquilibriumOptimizer` does multiple passes on the Apply nodes from the graph, trying to apply local and global optimizations. Conceptually, it tries to execute all global optimizations, and to apply all local optimizations on all nodes in the graph. If no optimization got applied during a pass, it stops. So it tries to find an equilibrium state where none of the optimizations get applied. This is useful when we do not know a fixed order for the execution of the optimization.
- `time 0.751s for 14 passes` means that it took .7s and did 14 passes over the graph.
- `nb nodes (start, end, max) 108 81 117` means that at the start, the graph had 108 node, at the end, it had 81 and the maximum size was 117.

- Then it prints some global timing information: it spent 0.029s in `io_toposort`, all local optimizers took 0.687s together for all passes, and global optimizers took a total of 0.010s.
- Then we print the timing for each pass, the optimization that got applied, and the number of time they got applied. For example, in pass 0, the `local_dimshuffle_lift` optimizer changed the graph 9 time.
- Then we print the time spent in each optimizer, the number of times they changed the graph and the number of nodes they introduced in the graph.
- Optimizations with that pattern *local_op_lift* means that a node with that op will be replaced by another node, with the same op, but will do computation closer to the inputs of the graph. For instance, `local_op(f(x))` getting replaced by `f(local_op(x))`.
- Optimization with that pattern *local_op_sink* is the opposite of *lift*. For instance `f(local_op(x))` getting replaced by `local_op(f(x))`.
- Local optimizers can replace any arbitrary node in the graph, not only the node it received as input. For this, it must return a dict. The keys being nodes to replace and the values being the corresponding replacement.

  This is useful to replace a client of the node received as parameter.

Tips

Reusing outputs

WRITEME

Don't define new Ops unless you have to

It is usually not useful to define Ops that can be easily implemented using other already existing Ops. For example, instead of writing a "sum_square_difference" Op, you should probably just write a simple function:

```
from theano import tensor as T

def sum_square_difference(a, b):
    return T.sum((a - b)**2)
```

Even without taking Theano's optimizations into account, it is likely to work just as well as a custom implementation. It also supports all data types, tensors of all dimensions as well as broadcasting, whereas a custom implementation would probably only bother to support contiguous vectors/matrices of doubles...

Use Theano's high order Ops when applicable

Theano provides some generic Op classes which allow you to generate a lot of Ops at a lesser effort. For instance, Elemwise can be used to make *elementwise* operations easily whereas DimShuffle can be used to

make transpose-like transformations. These higher order Ops are mostly Tensor-related, as this is Theano's specialty.

Op Checklist

Use this list to make sure you haven't forgotten anything when defining a new Op. It might not be exhaustive but it covers a lot of common mistakes.

WRITEME

Unit Testing

Theano relies heavily on unit testing. Its importance cannot be stressed enough!

Unit Testing revolves around the following principles:

- ensuring correctness: making sure that your Op, Type or Optimization works in the way you intended it to work. It is important for this testing to be as thorough as possible: test not only the obvious cases, but more importantly the corner cases which are more likely to trigger bugs down the line.
- test all possible failure paths. This means testing that your code fails in the appropriate manner, by raising the correct errors when in certain situations.
- sanity check: making sure that everything still runs after you've done your modification. If your changes cause unit tests to start failing, it could be that you've changed an API on which other users rely on. It is therefore your responsibility to either a) provide the fix or b) inform the author of your changes and coordinate with that person to produce a fix. If this sounds like too much of a burden... then good! APIs aren't meant to be changed on a whim!

This page is in no way meant to replace tutorials on Python's unittest module, for this we refer the reader to the official documentation. We will however adress certain specificities about how unittests relate to theano.

Unittest Primer

A unittest is a subclass of `unittest.TestCase`, with member functions with names that start with the string `test`. For example:

```
class MyTestCase(unittest.TestCase):
    def test0(self):
        pass
        # test passes cleanly

    def test1(self):
        self.assertTrue(2+2 == 5)
        # raises an exception, causes test to fail

    def test2(self):
        assert 2+2 == 5
        # causes error in test (basically a failure, but counted separately)
```

```
    def test2(self):
        assert 2+2 == 4
        # this test has the same name as a previous one,
        # so this is the one that runs.
```

How to Run Unit Tests ? Two options are available:

theano-nose The easiest by far is to use `theano-nose` which is a command line utility that recurses through a given directory, finds all unittests matching a specific criteria and executes them. By default, it will find & execute tests case in test*.py files whose method name starts with 'test'.

`theano-nose` is a wrapper around nosetests. You should be able to execute it if you installed Theano using pip, or if you ran "python setup.py develop" after the installation. If `theano-nose` is not found by your shell, you will need to add `Theano/bin` to your `PATH` environment variable.

Note: In Theano versions <= 0.5, `theano-nose` was not included. If you are working with such a version, you can call `nosetests` instead of `theano-nose` in all the examples below.

Running all unit tests

```
cd Theano/theano
theano-nose
```

Running unit tests with standard out

```
theano-nose -s
```

Running unit tests contained in a specific .py file

```
theano-nose <filename>.py
```

Running a specific unit test

```
theano-nose <filename>.py:<classname>.<method_name>
```

Using unittest module To launch tests cases from within python, you can also use the functionality offered by the `unittest` module. The simplest thing is to run all the tests in a file using `unittest.main()`. Python's built-in unittest module uses metaclasses to know about all the `unittest.TestCase` classes you have created. This call will run them all, printing '.' for passed tests, and a stack trace for exceptions. The standard footer code in theano's test files is:

```
if __name__ == '__main__':
    unittest.main()
```

You can also choose to run a subset of the full test suite.

To run all the tests in one or more `TestCase` subclasses:

```
suite = unittest.TestLoader()
suite = suite.loadTestsFromTestCase(MyTestCase0)
suite = suite.loadTestsFromTestCase(MyTestCase1)
...
unittest.TextTestRunner(verbosity=2).run(suite)
```

To run just a single `MyTestCase` member test function called `test0`:

```
MyTestCase('test0').debug()
```

Folder Layout "tests" directories are scattered throughout theano. Each tests subfolder is meant to contain the unittests which validate the .py files in the parent folder.

Files containing unittests should be prefixed with the word "test".

Optimally every python module should have a unittest file associated with it, as shown below. Unittests testing functionality of module <module>.py should therefore be stored in tests/test_<module>.py:

```
Theano/theano/tensor/basic.py
Theano/theano/tensor/elemwise.py
Theano/theano/tensor/tests/test_basic.py
Theano/theano/tensor/tests/test_elemwise.py
```

How to Write a Unittest

Test Cases and Methods Unittests should be grouped "logically" into test cases, which are meant to group all unittests operating on the same element and/or concept. Test cases are implemented as Python classes which inherit from unittest.TestCase

Test cases contain multiple test methods. These should be prefixed with the word "test".

Test methods should be as specific as possible and cover a particular aspect of the problem. For example, when testing the TensorDot Op, one test method could check for validity, while another could verify that the proper errors are raised when inputs have invalid dimensions.

Test method names should be as explicit as possible, so that users can see at first glance, what functionality is being tested and what tests need to be added.

Example:

```
import unittest
class TestTensorDot(unittest.TestCase):
    def test_validity(self):
        # do stuff
        ...
    def test_invalid_dims(self):
        # do more stuff
        ...
```

Test cases can define a special setUp method, which will get called before each test method is executed. This is a good place to put functionality which is shared amongst all test methods in the test case (i.e initializing data, parameters, seeding random number generators – more on this later)

```
class TestTensorDot(unittest.TestCase):
    def setUp(self):
        # data which will be used in various test methods
        self.avals = numpy.array([[1,5,3],[2,4,1]])
        self.bvals = numpy.array([[2,3,1,8],[4,2,1,1],[1,4,8,5]])
```

Similarly, test cases can define a tearDown method, which will be implicitely called at the end of each test method.

Checking for correctness When checking for correctness of mathematical expressions, the user should preferably compare theano's output to the equivalent numpy implementation.

Example:

```
class TestTensorDot(unittest.TestCase):
    def setUp(self):
        ...

    def test_validity(self):
        a = T.dmatrix('a')
        b = T.dmatrix('b')
        c = T.dot(a,b)
        f = theano.function([a,b],[c])
        cmp = f(self.avals,self.bvals) == numpy.dot(self.avals,self.bvals)
        self.assertTrue(numpy.all(cmp))
```

Avoid hard-coding variables, as in the following case:

```
self.assertTrue(numpy.all(f(self.avals,self.bvals)==numpy.array([[25,25,30,28],[21,18,14,25
```

This makes the test case less manageable and forces the user to update the variables each time the input is changed or possibly when the module being tested changes (after a bug fix for example). It also constrains the test case to specific input/output data pairs. The section on random values covers why this might not be such a good idea.

Here is a list of useful functions, as defined by TestCase:

- checking the state of boolean variables: assert, assertTrue, assertFalse
- checking for (in)equality constraints: assertEqual, assertNotEqual
- checking for (in)equality constraints up to a given precision (very useful in theano): assertAlmostEqual, assertNotAlmostEqual

Checking for errors On top of verifying that your code provides the correct output, it is equally important to test that it fails in the appropriate manner, raising the appropriate exceptions, etc. Silent failures are deadly, as they can go unnoticed for a long time and a hard to detect "after-the-fact".

Example:

```
class TestTensorDot(unittest.TestCase):
    ...
    def test_3D_dot_fail(self):
```

```
        def func():
            a = T.TensorType('float64', (False,False,False)) # create 3d tensor
            b = T.dmatrix()
            c = T.dot(a,b) # we expect this to fail
        # above should fail as dot operates on 2D tensors only
        self.assertRaises(TypeError, func)
```

Useful function, as defined by TestCase:

- assertRaises

Test Cases and Theano Modes When compiling theano functions or modules, a mode parameter can be given to specify which linker and optimizer to use.

Example:

```
f = T.function([a,b],[c],mode='FAST_RUN')
```

Whenever possible, unit tests should omit this parameter. Leaving out the mode will ensure that unit tests use the default mode. This default mode is set to the configuration variable `config.mode`, which defaults to 'FAST_RUN', and can be set by various mechanisms (see `config`).

In particular, the enviromnment variable `THEANO_FLAGS` allows the user to easily switch the mode in which unittests are run. For example to run all tests in all modes from a BASH script, type this:

```
THEANO_FLAGS='mode=FAST_COMPILE' theano-nose
THEANO_FLAGS='mode=FAST_RUN' theano-nose
THEANO_FLAGS='mode=DebugMode' theano-nose
```

Using Random Values in Test Cases numpy.random is often used in unit tests to initialize large data structures, for use as inputs to the function or module being tested. When doing this, it is imperative that the random number generator be seeded at the be beginning of each unit test. This will ensure that unittest behaviour is consistent from one execution to another (i.e always pass or always fail).

Instead of using `numpy.random.seed` to do this, we encourage users to do the following:

```
from theano.tests import unittest_tools

class TestTensorDot(unittest.TestCase):
    def setUp(self):
        unittest_tools.seed_rng()
        # OR ... call with an explicit seed
        unittest_tools.seed_rng(234234) #use only if really necessary!
```

The behaviour of seed_rng is as follows:

- If an explicit seed is given, it will be used for seeding numpy's rng.
- If not, it will use `config.unittests.rseed` (its default value is 666).
- If config.unittests.rseed is set to "random", it will seed the rng with None, which is equivalent to seeding with a random seed.

The main advantage of using unittest_tools.seed_rng is that it allows us to change the seed used in the unitests, without having to manually edit all the files. For example, this allows the nightly build to run theano-nose repeatedly, changing the seed on every run (hence achieving a higher confidence that the variables are correct), while still making sure unittests are deterministic.

Users who prefer their unittests to be random (when run on their local machine) can simply set `config.unittests.rseed` to 'random' (see `config`).

Similarly, to provide a seed to numpy.random.RandomState, simply use:

```
rng = numpy.random.RandomState(unittest_tools.fetch_seed())
# OR providing an explicit seed
rng = numpy.random.RandomState(unittest_tools.fetch_seed(1231)) #again not recommended
```

Note that the ability to change the seed from one nosetest to another, is incompatible with the method of hard-coding the baseline variables (against which we compare the theano outputs). These must then be determined "algorithmically". Although this represents more work, the test suite will be better because of it.

Creating an Op UnitTest

A few tools have been developed to help automate the development of unitests for Theano Ops.

Validating the Gradient The `verify_grad` function can be used to validate that the `grad` function of your Op is properly implemented. `verify_grad` is based on the Finite Difference Method where the derivative of function `f` at point `x` is approximated as:

$$\frac{\partial f}{\partial x} = lim_{\Delta \to 0} \frac{f(x+\Delta) - f(x-\Delta)}{2\Delta}$$

`verify_grad` performs the following steps:

- approximates the gradient numerically using the Finite Difference Method
- calculate the gradient using the symbolic expression provided in the `grad` function
- compares the two values. The tests passes if they are equal to within a certain tolerance.

Here is the prototype for the verify_grad function.

```
>>> def verify_grad(fun, pt, n_tests=2, rng=None, eps=1.0e-7, abs_tol=0.0001, rel_tol=0.00(
```

`verify_grad` raises an Exception if the difference between the analytic gradient and numerical gradient (computed through the Finite Difference Method) of a random projection of the fun's output to a scalar exceeds both the given absolute and relative tolerances.

The parameters are as follows:

- `fun`: a Python function that takes Theano variables as inputs, and returns a Theano variable. For instance, an Op instance with a single output is such a function. It can also be a Python function that calls an op with some of its inputs being fixed to specific values, or that combine multiple ops.
- `pt`: the list of numpy.ndarrays to use as input values

- `n_tests`: number of times to run the test
- `rng`: random number generator used to generate a random vector u, we check the gradient of sum(u*fn) at pt
- `eps`: stepsize used in the Finite Difference Method
- `abs_tol`: absolute tolerance used as threshold for gradient comparison
- `rel_tol`: relative tolerance used as threshold for gradient comparison

In the general case, you can define `fun` as you want, as long as it takes as inputs Theano symbolic variables and returns a sinble Theano symbolic variable:

```
def test_verify_exprgrad():
    def fun(x,y,z):
        return (x + tensor.cos(y)) / (4 * z)**2

    x_val = numpy.asarray([[1], [1.1], [1.2]])
    y_val = numpy.asarray([0.1, 0.2])
    z_val = numpy.asarray(2)
    rng = numpy.random.RandomState(42)

    tensor.verify_grad(fun, [x_val, y_val, z_val], rng=rng)
```

Here is an example showing how to use `verify_grad` on an Op instance:

```
def test_flatten_outdimNone():
    # Testing gradient w.r.t. all inputs of an op (in this example the op
    # being used is Flatten(), which takes a single input).
    a_val = numpy.asarray([[0,1,2],[3,4,5]], dtype='float64')
    rng = numpy.random.RandomState(42)
    tensor.verify_grad(tensor.Flatten(), [a_val], rng=rng)
```

Here is another example, showing how to verify the gradient w.r.t. a subset of an Op's inputs. This is useful in particular when the gradient w.r.t. some of the inputs cannot be computed by finite difference (e.g. for discrete inputs), which would cause `verify_grad` to crash.

```
def test_crossentropy_softmax_grad():
    op = tensor.nnet.crossentropy_softmax_argmax_1hot_with_bias
    def op_with_fixed_y_idx(x, b):
        # Input `y_idx` of this Op takes integer values, so we fix them
        # to some constant array.
        # Although this op has multiple outputs, we can return only one.
        # Here, we return the first output only.
        return op(x, b, y_idx=numpy.asarray([0, 2]))[0]

    x_val = numpy.asarray([[-1, 0, 1], [3, 2, 1]], dtype='float64')
    b_val = numpy.asarray([1, 2, 3], dtype='float64')
    rng = numpy.random.RandomState(42)

    tensor.verify_grad(op_with_fixed_y_idx, [x_val, b_val], rng=rng)
```

Note: Although `verify_grad` is defined in `theano.tensor.basic`, unittests should use the

version of `verify_grad` defined in `theano.tests.unittest_tools`. This is simply a wrapper function which takes care of seeding the random number generator appropriately before calling `theano.tensor.basic.verify_grad`

---

makeTester and makeBroadcastTester

Most Op unittests perform the same function. All such tests must verify that the op generates the proper output, that the gradient is valid, that the Op fails in known/expected ways. Because so much of this is common, two helper functions exists to make your lives easier: `makeTester` and `makeBroadcastTester` (defined in module `theano.tensor.tests.test_basic`).

Here is an example of `makeTester` generating testcases for the Dot product op:

```
DotTester = makeTester(name = 'DotTester',
                       op = dot,
                       expected = lambda x, y: numpy.dot(x, y),
                       checks = {},
                       good = dict(correct1 = (rand(5, 7), rand(7, 5)),
                                   correct2 = (rand(5, 7), rand(7, 9)),
                                   correct3 = (rand(5, 7), rand(7))),
                       bad_build = dict(),
                       bad_runtime = dict(bad1 = (rand(5, 7), rand(5, 7)),
                                          bad2 = (rand(5, 7), rand(8,3))),
                       grad = dict())
```

In the above example, we provide a name and a reference to the op we want to test. We then provide in the `expected` field, a function which `makeTester` can use to compute the correct values. The following five parameters are dictionaries which contain:

- checks: dictionary of validation functions (dictionary key is a description of what each function does). Each function accepts two parameters and performs some sort of validation check on each op-input/op-output value pairs. If the function returns False, an Exception is raised containing the check's description.
- good: contains valid input values, for which the output should match the expected output. Unittest will fail if this is not the case.
- bad_build: invalid parameters which should generate an Exception when attempting to build the graph (call to `make_node` should fail). Fails unless an Exception is raised.
- bad_runtime: invalid parameters which should generate an Exception at runtime, when trying to compute the actual output values (call to `perform` should fail). Fails unless an Exception is raised.
- grad: dictionary containing input values which will be used in the call to `verify_grad`

`makeBroadcastTester` is a wrapper function for makeTester. If an `inplace=True` parameter is passed to it, it will take care of adding an entry to the `checks` dictionary. This check will ensure that inputs and outputs are equal, after the Op's perform function has been applied.

Extending Theano: FAQ and Troubleshooting

I wrote a new Op/Type, and weird stuff is happening...

First, check the *Op's contract* and the *Type's contract* and make sure you're following the rules. Then try running your program in *Using DebugMode*. DebugMode might catch something that you're not seeing.

I wrote a new optimization, but it's not getting used...

Remember that you have to register optimizations with the *The optimization database (optdb)* for them to get used by the normal modes like FAST_COMPILE, FAST_RUN, and DebugMode.

I wrote a new optimization, and it changed my results even though I'm pretty sure it is correct.

First, check the *Op's contract* and make sure you're following the rules. Then try running your program in *Using DebugMode*. DebugMode might catch something that you're not seeing.

6.2.8 Developer Start Guide

Resources

See *Community* for a list of Theano resources. The following groups/mailing-lists are especially useful to Theano contributors: theano-dev, theano-buildbot, and theano-github.

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

Installation and configuration

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

This will create your own Theano project on GitHub, referred later as "YourProfile/Theano", or "origin", from which you will be able to contribute to the original Theano/Theano, also called "central".

Create a local copy

Clone your fork locally with

```
git clone git@github.com:YOUR_GITHUB_LOGIN/Theano.git
```

For this URL to work, you must set your public ssh keys inside your github account setting.

From your local repository, your own fork on GitHub will be called "origin".

Then, add a reference to the original ("central") Theano repository with

```
git remote add central git://github.com/Theano/Theano.git
```

You can choose another name than "central" to reference Theano/Theano (for instance, NumPy uses "upstream"), but this documentation will stick to "central."

You can then test your installation of Theano by following the steps of *Testing your installation*.

Using your local copy

To update your library to the latest revision, you should have a local branch that tracks central/master. You can add one (named "trunk" here) 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 Theano, you should work in another branch, like described in the *Development Workflow* section below.

Configure Git

On your local machine, you need to configure git with basic informations:

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

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

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

Development Workflow

Start a new local branch

When working on a new feature in your own fork, start from an up-to-date copy of the *master* branch (the principal one) of the central repository (Theano/Theano on GitHub):

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

Note: This last line is a shortcut for:

```
git branch my_shiny_feature central/master
git checkout my_shiny_feature
```

Submit your changes to the central repository

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

```
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. This will signal the maintainers that you wish to submit your changes for inclusion in central/master. If you don't get any feedback, bug us on the theano-dev mailing list.

Address reviewer comments

Your pull request will be reviewed by members of the core development team. If your branch is not directly accepted, the reviewers will use GitHub's system to add "notes", either general (on the entire commit), or "line notes", relative to a particular line of code. In order to have the pull request accepted, you may have to answer the reviewer's questions, you can do that on GitHub.

You may also have to edit your code to address their concerns. Some of the usual requests include fixing typos in comments, adding or correcting comments, adding unit tests in the test suite. In order to do that, you should continue your edits in the same branch you used (in this example, "my_shiny_feature"). For instance, if you changed your working branch, you should first:

```
git checkout my_shiny_feature
```

Then, edit your code, and test it appropriately (see *Tips for Quality Contributions* below), and push it again to your GitHub fork, like the first time (except the `-u` option is only needed the first time):

```
git push origin my_shiny_feature
```

The pull request to the central repository will then be automatically updated by GitHub. However, the reviewers will not be automatically notified of your revision, so it is advised to reply to the comments on GitHub, to let them know that you have submitted a fix.

Tips for Quality Contributions

Coding Style Auto Check

In Theano, we use the same coding style as the Pylearn project, except that we don't use the numpy docstring standard. The principal thing to know is that we follow the PEP 8 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 Theano 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 Theano and how to enable this option.

Setting up your Editor for PEP8

Here are instructions for *Vim* and *Emacs*. If you have similar instructions for other text editors or IDE, please let us know and we will update this documentation.

Vim Detection of warnings and errors is done by the pep8 script (or flake8, that also checks for other things, like syntax errors). Syntax highlighting and general integration into Vim is done by the Syntastic plugin for Vim.

To install flake8, simply run:

```
pip install flake8
```

You can use `easy_install` instead of `pip`, and `pep8` instead of `flake8` if you prefer. The important thing is that the `flake8` or `pep8` executable ends up in your `$PATH`.

To install Syntastic, according to its documentation, the easiest way is to install pathogen.vim first.

Here's a relevant extract of pathogen.vim's installation instructions:

> Install to `~/.vim/autoload/pathogen.vim`. Or copy and paste:
>
> ```
> mkdir -p ~/.vim/autoload ~/.vim/bundle; \
> curl -so ~/.vim/autoload/pathogen.vim \
>     https://raw.github.com/tpope/vim-pathogen/HEAD/autoload/pathogen.vim
> ```
>
> If you don't have `curl`, use `wget -O` instead.
>
> By the way, if you're using Windows, change all occurrences of `~/.vim` to `~\vimfiles`.
>
> Add this to your vimrc:
>
> ```
> call pathogen#infect()
> ```
>
> Now any plugins you wish to install can be extracted to a subdirectory under `~/.vim/bundle`, and they will be added to the `'runtimepath'`.

Now, we can install Syntastic. From the installation instructions:

```
cd ~/.vim/bundle
git clone https://github.com/scrooloose/syntastic.git
```

Then reload vim, run `:Helptags`, and check out `:help syntastic.txt`.

From now on, when you save into a Python file, a syntax check will be run, and results will be displayed using Vim's quickfix mechanism (more precisely, a location-list). A few useful commands are:

- Open the list of errors: `:lopen`, that can be abbreviated in `:lop` (denoted `:lop[en]`).
- Close that list: `:lcl[ose]`.
- Next error: `:lne[xt]`.
- Previous error: `:lp[revious]`.

Once you fix errors, messages and highlighting will still appear in the fixed file until you save it again.

We can also configure the `~/.vimrc` to make it easier to work with Syntastic. For instance, to add a summary in the status bar, you can add:

```
set statusline+=%{SyntasticStatuslineFlag()}
```

To bind F2 and F3 to navigate to previous and next error, you can add:

```
map <F2> :lprevious<CR>
map <F3> :lnext<CR>
```

You can prefix those by `autocmd FileType python` if you want these bindings to work only on Python files.

Emacs There is an **excellent** system to configure emacs for Python: emacs-for-python. It gathers many emacs config into one, and modifies them to behave together nicely. You can use it to check for pep8 compliance and for Python syntax errors.

To install it on Linux, you can do like this:

```
cd
git clone https://github.com/gabrielelanaro/emacs-for-python.git ~/.emacs.d/emacs-for-pytho
```

Then in your `~/.emacs` file, add this:

```
;; Mandatory
(load-file "~/.emacs.d/emacs-for-python/epy-init.el")
(add-to-list 'load-path "~/.emacs.d/emacs-for-python/") ;; tell where to load the various

;; Each of them enables different parts of the system.
;; Only the first two are needed for pep8, syntax check.
(require 'epy-setup) ;; It will setup other loads, it is required!
(require 'epy-python) ;; If you want the python facilities [optional]
(require 'epy-completion) ;; If you want the autocompletion settings [optional]
(require 'epy-editing) ;; For configurations related to editing [optional]
;; [newer version of emacs-for-python]
```

```
(require 'epy-nose) ;; For shortcut to call nosetests [optional]

;; Define f10 to previous error
;; Define f11 to next error
(require 'epy-bindings) ;; For my suggested keybindings [optional]

;; Some shortcut that do not collide with gnome-terminal,
;; otherwise, "epy-bindings" define f10 and f11 for them.
(global-set-key [f2] 'flymake-goto-prev-error)
(global-set-key [f3] 'flymake-goto-next-error)

;; Next two lines are the checks to do. You can add more if you wish.
(epy-setup-checker "pyflakes %f") ;; For python syntax check
(epy-setup-checker "pep8 -r %f") ;; For pep8 check
```

Note: The script highlights problematic lines. This can make part of the line not readable depending on the background. To replace the line highlight by an underline, add this to your emacs configuration file:

;; Make lines readable when there is an warning [optional] (custom-set-faces '(flymake-errline ((((class color)) (:underline "red")))) '(flymake-warnline ((((class color)) (:underline "yellow")))))

Unit tests

When you submit a pull request, your changes will automatically be tested via Travis-CI. This will post the results of the tests with a little icon next to your commit. A yellow circle means the tests are running. A red X means the tests failed and a green circle means the tests passed.

Just because the tests run automatically does not mean you shouldn't run them yourself to make sure everything is all right. You can run only the portion you are modifying to go faster and have travis to make sure there are no global impacts.

Also, if you are changing GPU code, travis doesn't test that, because there are no GPUs on the test nodes.

To run the test suite with the default options, you can follow the instructions of *Testing your installation*.

Each night we execute all the unit tests automatically, with several sets of options. The result is sent by email to the theano-buildbot mailing list.

For more detail, see *The nightly build/tests process*.

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

```
theano/misc/do_nightly_build
```

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

More Advanced Git Usage

You can find information and tips in the numpy development page. Here are a few.

Cleaning up branches

When your pull request has been merged, you can delete the branch from your GitHub fork's list of branches. This is useful to avoid having too many branches staying there. Deleting this remote branch is achieved with:

```
git push origin :my_shiny_feature
```

This lines pushes to the "origin" repository (your fork of Theano on GitHub), into the branch "my_shiny_feature", an empty content (that's why there is nothing before the colon), effectively removing it.

The branch will still be present in your local clone of the repository. If you want to delete it from there, too, you can run:

```
git branch -d my_shiny_feature
```

Amending a submitted pull request

If you want to fix a commit already submitted within a pull request (e.g. to fix a small typo), before the pull request is accepted, you can do it like this to keep history clean:

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

Do not abuse that command, and please use it only when there are only small issues to be taken care of. Otherwise, it becomes difficult to match the comments made by reviewers with the new modifications. In the general case, you should stick with the approach described above.

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.

Add another distant repository

To collaborate with another user on some feature he is developing, and that is not ready for inclusion in central, the easiest way is to use a branch of their Theano fork (usually on GitHub).

Just like we added Theano/Theano as a remote repository, named "central", you can add (on your local machine) a reference to their fork as a new remote repository. REPO_NAME is the name you choose to name this fork, and GIT_REPO_PATH is the URL of the fork in question.

```
git remote add REPO_NAME GIT_REPO_PATH
```

Then, you can create a new local branch (LOCAL_BRANCH_NAME) based on a specific branch (REMOTE_BRANCH_NAME) from the remote repository (REPO_NAME):

```
git checkout -b LOCAL_BRANCH_NAME REPO_NAME/REMOTE_BRANCH_NAME
```

Other tools that can help you

- cProfile: time profiler that work at function level.
- Yep: A module for profiling compiled extensions.
- autopep8: A tool that automatically formats Python code to conform to the PEP 8 style guide.
- line_profiler: Line-by-line profiler.
- memory_profiler: memory profiler
- runsnake: Gui for cProfile(time profiler) and Meliae(memory profiler)
- Guppy: Supports object and heap memory sizing, profiling and debugging.
- hub: A tool that adds github commands to the git command line.
- git pull-requests: Another tool for git/github command line.

6.2.9 Glossary

Apply Instances of `Apply` represent the application of an *Op* to some input *Variable* (or variables) to produce some output *Variable* (or variables). They are like the application of a [symbolic] mathematical function to some [symbolic] inputs.

Broadcasting Broadcasting is a mechanism which allows tensors with different numbers of dimensions to be used in element-by-element (elementwise) computations. It works by (virtually) replicating the smaller tensor along the dimensions that it is lacking.

For more detail, see *Broadcasting in Theano vs. Numpy*, and also * SciPy documentation about numpy's broadcasting * OnLamp article about numpy's broadcasting

Constant A variable with an immutable value. For example, when you type

```
>>> x = tensor.ivector()
>>> y = x + 3
```

Then a *constant* is created to represent the `3` in the graph.

See also: `gof.Constant`

Elementwise An elementwise operation `f` on two tensor variables `M` and `N` is one such that:

```
f(M, N)[i, j] == f(M[i, j], N[i, j])
```

In other words, each element of an input matrix is combined with the corresponding element of the other(s). There are no dependencies between elements whose `[i, j]` coordinates do not correspond, so an elementwise operation is like a scalar operation generalized along several dimensions. Elementwise operations are defined for tensors of different numbers of dimensions by *broadcasting* the smaller ones.

Expression See *Apply*

Expression Graph A directed, acyclic set of connected *Variable* and *Apply* nodes that express symbolic functional relationship between variables. You use Theano by defining expression graphs, and then compiling them with *theano.function*.

See also *Variable*, *Op*, *Apply*, and *Type*, or read more about *Graph Structures*.

Destructive An *Op* is destructive (of particular input[s]) if its computation requires that one or more inputs be overwritten or otherwise invalidated. For example, *inplace* Ops are destructive. Destructive Ops can sometimes be faster than non-destructive alternatives. Theano encourages users not to put destructive Ops into graphs that are given to *theano.function*, but instead to trust the optimizations to insert destructive ops judiciously.

Destructive Ops are indicated via a `destroy_map` Op attribute. (See `gof.Op`.

Graph see *expression graph*

Inplace Inplace computations are computations that destroy their inputs as a side-effect. For example, if you iterate over a matrix and double every element, this is an inplace operation because when you are done, the original input has been overwritten. Ops representing inplace computations are *destructive*, and by default these can only be inserted by optimizations, not user code.

Linker Part of a function *Mode* – an object responsible for 'running' the compiled function. Among other things, the linker determines whether computations are carried out with C or Python code.

Mode An object providing an *optimizer* and a *linker* that is passed to *theano.function*. It parametrizes how an expression graph is converted to a callable object.

Op The `.op` of an *Apply*, together with its symbolic inputs fully determines what kind of computation will be carried out for that `Apply` at run-time. Mathematical functions such as addition (`T.add`) and indexing `x[i]` are Ops in Theano. Much of the library documentation is devoted to describing the various Ops that are provided with Theano, but you can add more.

See also *Variable*, *Type*, and *Apply*, or read more about *Graph Structures*.

Optimizer An instance of `Optimizer`, which has the capacity to provide an *optimization* (or optimizations).

Optimization A *graph* transformation applied by an *optimizer* during the compilation of a *graph* by *theano.function*.

Pure An *Op* is *pure* if it has no *destructive* side-effects.

Storage The memory that is used to store the value of a Variable. In most cases storage is internal to a compiled function, but in some cases (such as *constant* and *shared variable* the storage is not internal.

Shared Variable A *Variable* whose value may be shared between multiple functions. See `shared` and `theano.function`.

theano.function The interface for Theano's compilation from symbolic expression graphs to callable objects. See `function.function()`.

Type The `.type` of a *Variable* indicates what kinds of values might be computed for it in a compiled graph. An instance that inherits from `Type`, and is used as the `.type` attribute of a *Variable*.

See also *Variable*, *Op*, and *Apply*, or read more about *Graph Structures*.

Variable The the main data structure you work with when using Theano. For example,

```
>>> x = theano.tensor.ivector()
>>> y = -x**2
```

`x` and `y` are both *Variables*, i.e. instances of the `Variable` class.

See also *Type*, *Op*, and *Apply*, or read more about *Graph Structures*.

View Some Tensor Ops (such as Subtensor and Transpose) can be computed in constant time by simply reindexing their inputs. The outputs from [the Apply instances from] such Ops are called *Views* because their storage might be aliased to the storage of other variables (the inputs of the Apply). It is important for Theano to know which Variables are views of which other ones in order to introduce *Destructive* Ops correctly.

View Ops are indicated via a `view_map` Op attribute. (See `gof.Op`.

6.2.10 Links

This page lists links to various resources.

Theano requirements

- git: A distributed revision control system (RCS).
- nosetests: A system for unit tests.
- numpy: A library for efficient numerical computing.
- python: The programming language Theano is for.
- scipy: A library for scientific computing.

Libraries we might want to look at or use

This is a sort of memo for developers and would-be developers.

- autodiff: Tools for automatic differentiation.
- boost.python: An interoperability layer between Python and C++
- cython: A language to write C extensions to Python.
- liboil: A library for CPU-specific optimization.
- llvm: A low-level virtual machine we might want to use for compilation.

- networkx: A package to create and manipulate graph structures.
- pycppad: Python bindings to an AD package in C++.
- pypy: Optimizing compiler for Python in Python.
- shedskin: An experimental (restricted-)Python-to-C++ compiler.
- swig: An interoperability layer between Python and C/C++
- unpython: Python to C compiler.

6.2.11 Internal Documentation

If you're feeling ambitious, go fix some *pylint <http://lgcm.iro.umontreal.ca/auto_theano_pylint/pylint_global.html>* errors!

Release

Having a release system has many benefits. First and foremost, it makes trying out Theano easy. You can install a stable version of Theano, without having to worry about the current state of the repository. While we usually try NOT to break the trunk, mistakes can happen. This also greatly simplifies the installation process: mercurial is no longer required and certain python dependencies can be handled automatically (numpy for now, maybe pycuda, cython later).

The Theano release plan is detailed below. Comments and/or suggestions are welcome on the mailing list.

1. We will perform a monthly release of Theano. These will be "lightweight" releases and will include everything that was done in the last month. All outstanding feature requests are pushed back to the following month, so as not to delay the current release.
2. Asynchronous releases will only be made when a bug generating incorrect output is discovered and fixed.
3. Each release must satisfy the following criteria. Non-compliance will result in us delaying or skipping the release in question.
   (a) No regression errors.
   (b) No known, silent errors.
   (c) No errors giving incorrect results.
   (d) No test errors/failures, except for known errors.
      i. Known errors should not be used to encode "feature wish lists", as is currently the case.
      ii. Incorrect results should raise errors and not known errors (this has always been the case)
      iii. All known errors should have a ticket and a reference to that ticket in the error message.
   (e) All commits should have been reviewed, to ensure none of the above problems are introduced.
4. The release numbers will follow the X.Y.Z scheme:
   (a) We update Z by 1 for each lightweight release.

(b) We update Y for bug fixes, interface changes and/or significant features we wish to publicize.

(c) The Theano v1.0.0 release will be made when the interface is deemed stable enough and covers most of numpy's interface.

5. The trunk will be tagged on each release.
6. Each release will be uploaded to pypi.python.org, mloss.org and freshmeat.net
7. Release emails will be sent to theano-users, theano-announce, numpy-discussion@scipy.org and scipy-user@scipy.org .

Optional:

8. A 1-week scrum might take place before a release, in order to fix bugs which would otherwise prevent a release.

(a) Occasional deadlines might cause us to skip a release.

(b) Everybody can (and should) participate, even people on the mailing list.

(c) The scrum should encourage people to finish what they have already started (missing documentation, missing test, ...). This should help push out new features and keep the documentation up to date.

(d) If possible, aim for the inclusion of one new interesting feature.

(e) Participating in the scrum should benefit all those involved, as you will learn more about our tools and help develop them in the process. A good indication that you should participate is if you have a need for a feature which is not yet implemented.

Developer Start Guide MOVED!

The developer start guide *moved*.

LISA Labo specific instructions

Tips for running at LISA

Shell configuration files `/opt/lisa/os/.local.{bash,csh}rc` should define `THEANORC` to include `/opt/lisa/os/.local.theanorc` as a configuration file.

`/opt/lisa/os/.local.theanorc` should include the right default values for the lab, in particular, `blas.ldflags` should contain '-lgoto'.

Tips for running on a cluster

Running Theano on Mammouth For instructions on running Theano on the mammouth cluster.

Running Theano on Mammouth

To run Theano on the Mammouth cluster, follow these simple steps:

- Make sure to source Fred's .local.bashrc file. It contains all the goodies for using the latest and greatest (optimized) libraries (numpy, scipy, etc.)

  ```
  >>> source /home/bastienf/.local.bashrc
  ```

  Perhaps even put this in your `.bashrc`

- set `config.blas.ldflags` to `'-lmkl -lguide -fopenmp'` (see `config` to know how)

  Note: the -lguide flag works, however the fix should probably be considered temporary. Intel has deprecated libguide.so in favor of the newer library libiomp5.so. However, both libraries are mutually exclusive and one component (theano, numpy or scipy?) already seems to be using libguide.so (hence -liomp5 causes a linking error when compiling thunks)

Documentation Documentation AKA Meta-Documentation

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 Theano/
python ./doc/scripts/docgen.py
```

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

```
cd Theano/
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
```

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)

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.

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.

How documentation is built on deeplearning.net

The server that hosts the theano documentation runs a cron job roughly every 2 hours that fetches a fresh Theano 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 theano-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).

pylint

pylint output is not autogenerated anymore.

Pylint documentation is generated using pylintrc file: `Theano/doc/pylintrc`

You can see a list of all pylint messages.

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 a script that download/update the repo of Theano, Pylearn, Pylearn2 and the Deep Learning Tutorial, then run their tests script under `*/misc/do_nightly_build`. Those script tests the project under various condition. The cron job also run some tests in Python 2.4 and Python 3.3 for Theano.

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

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.

Python booster

This page will give you a warm feeling in your stomach.

Non-Basic Python features

Theano doesn't use your grandfather's python.

- properties

  a specific attribute that has get and set methods which python automatically invokes.

  See [http://www.python.org/doc/newstyle/ New style classes].

- static methods vs. class methods vs. instance methods

- Decorators:

  ```
  @f
  def g():
     ...
  ```

  runs function `f` before each invocation of `g`. See PEP 0318. `staticmethod` is a specific decorator, since python 2.2

- `__metaclass__` is kinda like a decorator for classes. It runs the metaclass __init__ after the class is defined
- `setattr` + `getattr` + `hasattr`
- `*args` is a tuple like argv in C++, `**kwargs` is a keyword args version
- `pass` is no-op.
- functions (function objects) can have attributes too. This technique is often used to define a function's error messages.

```
def f(): return f.a
f.a = 5
f() # returns 5
```

- Warning about mutual imports:
  - script a.py file defined a class A.
  - script a.py imported file b.py
  - file b.py imported a, and instantiated a.A()
  - script a.py instantiated its own A(), and passed it to a function in b.py
  - that function saw its argument as being of type __main__.A, not a.A.

  Incidentally, this behaviour is one of the big reasons to put autotests in different files from the classes they test!

  If all the test cases were put into <file>.py directly, then during the test cases, all <file>.py classes instantiated by unit tests would have type `__main__.<classname>`, instead of type `<file>.<classname>`. This should never happen under normal usage, and can cause problems (like the one you are/were experiencing).

How to make a release

Update files

Update the NEWS.txt and move the old stuff in the HISTORY.txt file. To update the NEWS.txt file, check all ticket closed for this release and all commit log messages. Update the index.txt *News* section.

Update the "Vision"/"Vision State" in the file `Theano/doc/introduction.txt`.

Get a fresh copy of the repository

Clone the code:

```
git clone git@github.com:Theano/Theano.git Theano-0.X
```

It does not have to be in your PYTHONPATH.

Update the version number

Edit `setup.py` to contain the newest version number

```
cd Theano-0.X
vi setup.py     # Edit the MAJOR, MINOR, MICRO and SUFFIX
```

`conf.py` in the `doc/` directory should be updated in the following ways:

- Change the `version` and `release` variables to new version number.
- Change the upper copyright year to the current year if necessary.

Update the year in the `Theano/LICENSE.txt` file too, if necessary.

`NEWS.txt` usually contains the name and date of the release, change them too.

Update the code and the documentation for the theano flags `warn.ignore_bug_before` to accept the new version. You must modify the file `theano/configdefaults.py` and `doc/library/config.txt`.

Tag the release

You will need to commit the previous changes, tag the resulting version, and push that into the original repository. The syntax is something like the following:

```
git commit -m "Modifications for 0.X.Y release" setup.py doc/conf.py NEWS.txt HISTORY.txt t
git tag -a rel-0.X.Y
git push
git push --tags
```

The documentation will be automatically regenerated in the next few hours.

Generate and upload the package

For release candidates, only upload on PyPI.

On PyPI Now change `ISRELEASED` in `setup.py` to `True`.

Finally, use setuptools to register and upload the release:

```
python setup.py register sdist --formats=gztar,zip upload
```

This command register and uploads the package on pypi.python.org. To be able to do that, you must register on PyPI (you can create an new account, or use OpenID), and be listed among the "Package Index Owners" of Theano.

There is a bug in some versions of distutils that raises a UnicodeDecodeError if there are non-ASCII characters in NEWS.txt. You would need to change NEWS.txt so it contains only ASCII characters (the problem usually comes from diacritics in people's names).

On mloss.org Project page is at http://mloss.org/software/view/241/. Account jaberg is listed as submitter.

1. log in as jaberg to mloss
2. search for theano and click the logo
3. press 'update this project' on the left and change
   - the version number
   - the download link
   - the description of what has changed
4. press save

Make sure the "what's changed" text isn't too long because it will show up on the front page of mloss. You have to indent bullet lines by 4 spaces I think in the description.

You can "update this project" and save lots of times to get the revision text right. Just do not change the version number.

Finally Change `ISRELEASED` back to `False`.

Generate and upload the Windows installer

We are now able to build and distribute an MSI installer for Windows, assuming that Anaconda is the installed Python distribution. This installer is generated by WiX from an XML file, stored in the Theano-wininstaller Git repository.

- Install WiX if it is not already installed.
- On a Windows machine, checkout the `Theano-wininstaller` repository:

  ```
  git checkout https://github.com/Theano/Theano-wininstaller.git
  ```

- In `Theano-wininstaller\src`, create a *new* `theano_installer_<version>.wxs` from the previous one. We want to keep a history of these files, as they contain globally unique IDs.
- Change the strings and GUIDs appropriately, see the WiX tutorial for a reference.
- Compile the `.wxs` file following the instructions in it, it will be something like:

  ```
  candle.exe theano_installer_<version>.wxs
  light.exe -ext WixUIExtension theano_installer_<version>.wixobj
  ```

  This will generate a `theano_installer_<version>.msi` file in `src`.

- Test it by trying to install and uninstall it. It can be done by double-clicking on it, then uninstalling it from the Windows control panel, or (more easily) from the command line, which also allows to save the logs (use the `*v` modifier to increase verbosity):

  ```
  msiexec /i <file>.msi [/l[*v] install.log]
  msiexec /x <file>.msi [/l[*v] uninstall.log]
  ```

- When the test works, copy `theano_installer_<version>.msi` into `Theano-wininstaller\bin`, overwrite `bin\theano_installer_latest.msi` with another copy, then add the new files into the Git repository, and push to master:

```
copy src\theano_installer_<version>.msi bin\
copy /y src\theano_installer_<version>.msi bin\theano_installer_latest.msi
git add src\theano_installer_<version>.wxs
git add bin\theano_installer_<version>.msi
git add bin\theano_installer_latest.msi
git commit
git push
```

Announce the release

Generate an e-mail from the template in in `EMAIL.txt`, including content from `NEWS.txt`, and send it to the following mailing lists:

- theano-users
- theano-announce
- numpy-discussion@scipy.org
- scipy-user@scipy.org
- G+, Scientific Python: https://plus.google.com/communities/108773711053400791849

For release candidates, only e-mail:

- theano-announce
- theano-dev
- theano-users

6.2.12 Acknowledgements

- The developers of NumPy. Theano is based on its ndarray object and uses much of its implementation.
- The developers of SciPy. Our sparse matrix support uses their sparse matrix objects. We also reuse other parts.
- All Theano contributors.
- All Theano users that have given us feedback.
- The GPU implementation of tensordot is based on code from Tijmen Tieleman's gnumpy
- The original version of the function `cpuCount()` in the file *theano/misc/cpucount.py* come from the project pyprocessing. It is available under the same license as Theano.
- Our random number generator implementation on CPU and GPU uses the MRG31k3p algorithm that is described in:

16. L'Ecuyer and R. Touzin, Fast Combined Multiple Recursive Generators with Multipliers of the form a = +/- 2^d +/- 2^e, Proceedings of the 2000 Winter Simulation Conference, Dec. 2000, 683–689.

    We were authorized by Pierre L'Ecuyer to copy/modify his Java implementation in the SSJ software and to relicense it under BSD 3-Clauses in Theano.

- A better GPU memory allocator `CNMeM` is included in Theano. It has the same license.

6.2.13 LICENSE

U

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z