yann Documentation

Release 1.0rc1

Ragav Venkatesan

August 30, 2017

Contents

Welcome to the Yann Toolbox. It is a toolbox for building and learning convolutional neural networks, built on top of theano. This toolbox is a homage to Prof. Yann LeCun, one of the earliest poineers of CNNs. To setup the toolbox refer the *Installation Guide* guide. Once setup, you may start with the *Quick Start* guide or try your hand at the *Tutorials* and the guide to *Getting Started*. A user base discussion group is setup on gitter and also on google groups.

If you are here for the theano-tensorflow migration tool, click [here](http://www.tf-lenet.readthedocs.io).

Warning: Yann is currently under its early phases and is presently undergoing massive development. Expect a lot of changes. Unittests are only starting to be written, therefore the coverage and travis build passes are not to be completely trusted. The toolbox will be formalized in the future but at this moment, the authorship, coverage and maintanence of the toolbox is under extremely limited manpower.

Note: While, there are more formal and wholesome toolboxes that are similar and have a much larger userbase such as Lasagne, Keras, Blocks and Caffe, this toolbox is designed differently. This is much simpler and versatile. Yann is designed as a supplement to an upcoming beginner's book on Convolutional Neural Networks and also the toolbox of choice for a introductory course on deep learning for computer vision.

Because of this reason, Yann is specifically designed to be intuitive and easy to use for beginners. That does not compromise Yann of any of its core purpose - to be able to build CNNs in a plug and play fashion. It is still a good choice for a toolbox for running pre-trained models and build complicated, non-vannilla CNN architectures that are not easy to build with the other toolboxes. It is also a good choice for researchers and industrial scientists, who want to quickly prototype networks and test them before developing production scale models.

CHAPTER 1

Getting Started

The following will help you get quickly acquinted with Yann.

Installation Guide

Yann is built on top of Theano. Theano and all its pre-requisites are mandatory. Once theano and its pre-requisites are setup you may setup and run this toolbox. Theano setup is documented in the theano toolbox documentation. Yann is built with theanoo 0.8 but should be forward compatible unless theano makes a drastic release.

Quick fire Installation

Now before going through the full-fledged installation procedure, you can run through the entire installation in one command that will install the basics required to run the toolbox. To install the toolbox quickly do the following:

```
pip install git+git://github.com/ragavvenkatesan/yann.git
```

If it showed any errors, install `numpy` first. `skdata` has some issue that requires `numpy` installed first. If you use anaconda, just install the numpy and scipy using `conda install` instead of `pip install`. This will setup the toolbox for all intentions and purposes.

Verify that the installation of theano is indeed version 0.9 or greater by doing the following in a python shell

```
import theano
theano.__version__
```

If the version was not 0.9, you can install 0.9 by doing the following:

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

For a full-fledged installation procedure, don't do the above but run through the following set of instructions. If you want to install all other supporting features like datasets, visualizers and others, do the following:

```
pip install -r requirements_full.txt
pip install git+git://github.com/ragavvenkatesan/yann.git
```

Full installation

Dependencies

Python + pip / conda

Yann needs Python 2.7. Please install it for your OS.. Some modules that are required don't come with default python. But don't worry python comes with a package installer called pip. You can use pip to install additional packages.

For a headache free installation, the anaconda distribution of python is very strongly recommended because it comes with a lot of goodies pre-packaged.

C compiler

You need a C compiler, not because yann needs C, but theano and probably numpy requires C compilers. Make sure that your OS has one. Apple osX or macOS users, if you are using Cuda and cuDNN, prefer using command line tools 7.x+. 8 doesn't work with cuDNN at the moment of writing this documentation. You can download older versions of xcode and command line tools here.

numpy/scipy

Numpy 1.6 and Scipy 0.11 are needed for yann. Make sure these work well with a blas system. Prefer Intel MKL for blas, which is also availabe from anaconda. MKL is free for students and researchers and is available for a small price for others.

If you use pip use

```
pip install numpy
pip install scipy
```

to install these. If you use anaconda, use

```
conda install mkl
conda install numpy
conda install scipy
```

to set these up. If not, yann installer will `pip install numpy scipy` anyway as part of its requirements.

Theano

Once all the pre-requisites are setup, install theano version 0.8 or higher.

The following `.theanorc` configuration can be used as a sample normally, but you may choose other options. As an example one can use the following:

```
[global]
floatX=float32
device=cuda0
optimizer_including=cudnn
mode = FAST_RUN

[nvcc]
nvcc.fastmath=True
allow_gc=False

[cuda]
root=/usr/local/cuda/

[blas]
ldflags = -lmkl

[lib]
cnmem = 0.5
```

If you use the libgpuarray backend instead of the CUDA backend, use `device=cuda0` or whichever device you want to run on. If you are using CUDA backed use `device=gpu0`. Refer theano documentation for more on this.

Optional Dependencies

These are some optional dependencies that yann doesn't use directly but are used by yann's dependencies like theano. I highly recommend these before installing theano.

Cuda

This is an optional dependency. If you need the capability of a Nvidia GPU, you will need a suitable CUDA toolkit and drivers. If you do not have this dependency installed, you won't be able to run the code on Nvidia GPUs.Some compoenents of the code depend on cuDNN for speeding things up, so cuDNN is highly recommended although optional. Nvidia has the awesome cuDNN library that is free as long as you register as a developer. If you didn't install CUDA, you can still run the toolbox, but it will be much slower running on a CPU.

Libgpuarray

libgpuarray is now fully supported, cuda backend is strongly recommended for macOS, but for the Pascal architecture of GPUs, `libgpuarray` seems to be performing much better. This is also an optional but highly recommended tool

Additional Dependencies

Yann also needs the following as additional dependencies that opens up additional features.

Networkx

For those who are networking geeks, a neural network is a directed acyclic graph. So Yann internally has the ability for every network to create a `networkx` style graph and do things with it if you need.

Networkx is a tremendously popular tool for network realted tasks and we are still exploring and testing its capabilities. This might only ever be used for visualization of network purposes, but some researcher somewhere might use this once in the future networks get sophisticated, we never know. This is an optional dependency, not having this dependency doesn't affect the toolbox, except for the purposes it is needed for.

You can install `networkx` as follows:

```
pip install networkx
```

skdata

Used as a port for datasets. This is Needed if you are using some common benchmark datasets. Although this is an additional dependency, skdata is the core of the datasets module and most datasets in this toolbox are ported through skdata unless you have matlab. Work is on-going in integrating with fuel and other ports.

Install by using the following command:

```
pip install skdata
```

progressbar

Yann uses progressbar for aesthetic printing. You can install it easily by using

```
pip install progressbar
```

If you don't have progressbar, yann will simply ignore it and print progress on terminal.

Dependencies for visualization

Theano needs pydot and graphviz for visualization. We use theano's visualization for printing theano functions as shown here.

These visualizations are highly useful during debugging. If you want the capability of producing these for your networks, install the dependencises using the following commands:

```
apt-get install graphviz
pip install graphviz
pip install pydot pydot-ng
```

Not needed now, but might need in future. Yann will switch from openCV to matplotlib or browser matplotlib for visualization. Install it by

```
pip insall matplotlib
```

cPickle, gzip and hdf5py

Most often the case is that *cPickle* and *gzip* these come with the python installation, if not please install them. Yann uses these for saving down models and such.

For datasets, at the moment, yann uses cpickle. In the future, yann will migrate to hdf5 for datasets. We don't use hdf5py at the moment. Install hdf5py by running either,

```
conda install h5py
```

or,

```
pip install h5py
```

Yann Toolbox Setup

Finally to install the toolbox run,

```
pip install git+git://github.com/ragavvenkatesan/yann.git
```

If you have already setup the toolbox and want to just update to the bleeding-edge use,

```
pip install --upgrade git+git://github.com/ragavvenkatesan/yann.git
```

If you want to build by yourself you may clone from git and then run using setuptools. Ensure that you have setuptools installed first.

```
pip install git setuptools
```

Once you are done, you clone the repository from git.

```
git clone http://github.com/ragavvenkatesan/yann
```

Once cloned, enter the directory and run installer.

```
cd yann
python setup.py install
```

You can run a bunch of tests (working on it) by running the following code:

```
python setup.py test
```

Tutorials

If you are here for the first time you might want to consider doing the *Quick Start* instead of doing the tutorials. The tutorials are meant for those who have initial practice or experience with the toolbox and its structure. If you'd just want to see the codes or run the examples for testing or other such purposes you could follow this tutorial/API. I recommend going through the tutorial just in case though.

Logistic Regression.

Tutorial for logistic regression is basically the *Quick Start* guide. Please follow the tutorial there. A full working code is presented in the following.

Notes

This code contains one method that explains how to build a logistic regression classifier for the MNIST dataset using the yann toolbox.

For a more interactive tutorial refer the notebook at yann/pantry/tutorials/notebooks/Logistic Regression.ipynb

pantry.tutorials.log_reg.**log_reg**(*dataset*)

This function is a demo example of logistic regression.

Multi-layer Neural Network.

By virture of being here, it is assumed that you have gone through the *Quick Start*. Let us take this one step further and create a neural network with two hidden layers. We begin as usual by importing the network class and creating the input layer.

```
from yann.network import network
net = network()
dataset_params  = { "dataset": "_datasets/_dataset_xxxxxx", "id": 'mnist', "n_classes
↪" : 10 }
net.add_layer(type = "input", id ="input", dataset_init_args = dataset_params)
```

Instead of connecting this to a classfier as we saw in the *Quick Start* , let us add a couple of fully connected hidden layers. Hidden layers can be created using layer `type = dot_product`.

```
net.add_layer (type = "dot_product",
               origin ="input",
               id = "dot_product_1",
               num_neurons = 800,
               regularize = True,
               activation ='relu')

net.add_layer (type = "dot_product",
               origin ="dot_product_1",
               id = "dot_product_2",
               num_neurons = 800,
               regularize = True,
               activation ='relu')
```

Notice the parameters passed. `num_neurons` is the number of nodes in the layer. Notice also how we modularized the layers by using the `id` parameter. `origin` represents which layer will be the input to the new layer. By default yann assumes all layers are input serially and chooses the last added layer to be the input. Using `origin`, one can create various types of architectures. Infact any directed acyclic graphs (DAGs) that could be hand-drawn could be implemented. Let us now add a classifier and an objective layer to this.

```
net.add_layer ( type = "classifier",
                id = "softmax",
                origin = "dot_product_2",
                num_classes = 10,
                activation = 'softmax',
                )

net.add_layer ( type = "objective",
                id = "nll",
                origin = "softmax",
                )
```

Again notice that we have supplied a lot more arguments than before. Refer the API for more details. Let us create our own optimizer module this time instead of using the yann default. For any `module` in yann, the initialization can be done using the `add_module` method. The `add_module` method typically takes input `type` which in this case is `optimizer` and a set of intitliazation parameters which in our case is `params = optimizer_params`.

Any module params, which in this case is the `optimizer_params` is a dictionary of relevant options. A typical `optimizer setup` is:

```
optimizer_params =  {
            "momentum_type"       : 'polyak',
            "momentum_params"     : (0.9, 0.95, 30),
            "regularization"      : (0.0001, 0.0002),
            "optimizer_type"      : 'rmsprop',
            "id"                  : 'polyak-rms'
                    }
net.add_module ( type = 'optimizer', params = optimizer_params )
```

We have now successfully added a Polyak momentum with RmsProp back propagation with some L_1 and $L2$ coefficients that will be applied to the layers for which we passed as argument `regularize = True`. For more options of parameters on optimizer refer to the optimizer documentation . This optimizer will therefore solve the following error:

$$e(\mathbf{w_2}, \mathbf{w_1}, \mathbf{w}_\sigma) = \sigma(\mathbf{d_2}(\mathbf{d_1}(\mathbf{x}, \mathbf{w_1}), \mathbf{w_2})\mathbf{w}_\sigma) + \mathbf{0.0001}(|\mathbf{w_2}| + |\mathbf{w_1}| + |\mathbf{w}_\sigma|) + \mathbf{0.0002}(||\mathbf{w_2}|| + ||\mathbf{w_1}|| + ||\mathbf{w}_\sigma||)$$

where e is the error, $\sigma(.)$ is the sigmoid layer and $d_i(.)$ is the ith layer of the network. Once we are done, we can cook, train and test as usual:

```
learning_rates = (0.05, 0.01, 0.001)

net.cook( optimizer = 'polyak-rms',
          objective_layer = 'nll',
          datastream = 'mnist',
          classifier = 'softmax',
          )

net.train( epochs = (20, 20),
           validate_after_epochs = 2,
           training_accuracy = True,
           learning_rates = learning_rates,
           show_progress = True,
           early_terminate = True)
```

The `learning_rate`, supplied here is a tuple. The first indicates a annealing of a linear rate, the second is the initial learning rate of the first era, and the third value is the leanring rate of the second era. Accordingly, `epochs` takes in a tuple with number of epochs for each era.

This time, let us not let it run the forty epochs, let us cancel in the middle after some epochs by hitting ^c. Once it stops lets immediately test and demonstrate that the `net` retains the parameters as updated as possible. Once done, lets run `net.test()`.

Some new arguments are introduced here and they are for the most part easy to understand in context. `epoch` represents a `tuple` which is the number of epochs of training and number of epochs of fine tuning epochs after that. There could be several of these stages of finer tuning. Yann uses the term 'era' to represent each set of epochs running with one learning rate. `show_progress` will print a progress bar for each epoch. `validate_after_epochs` will perform validation after such many epochs on a different validation dataset. The full code for this tutorial with additional commentary can be found in the file `pantry.tutorials.mlp.py`. If you have toolbox cloned or downloaded or just the tutorials downloaded, Run the code as,

```
from pantry.tutorials.mlp import mlp
mlp(dataset = 'some dataset created')
```

or simply,

```
python pantry/tutorials/mlp.py
```

from the toolbox root or path added to toolbox. The __init__ program has all the required tools to create or load an already created dataset. Optionally as command line argument you can provide the location to the dataset.

pantry.tutorials.mlp.**mlp**(*dataset*, *verbose=1*)

This method is a tutorial on building a two layer multi-layer neural network. The built network is mnist->800->800->10 .It optimizes with polyak momentum and rmsprop.

Parameters **dataset** – an already created dataset.

Autoencoder Network.

By virture of being here, it is assumed that you have gone through the *Quick Start*.

Todo

Code is done, but text needs to be written in.

The full code for this tutorial with additional commentary can be found in the file pantry.tutorials.autoencoder.py. If you have toolbox cloned or downloaded or just the tutorials downloaded, Run the code as,

Todo

- Need a validation and testing thats better than just measuring rmse. Can't find something great.

Notes

This code contains two methods.

1. A shallow autoencoder with just one layer.
2. A Convolutional-Deconvolutional autoencoder that uses a deconv layer.

Both these methods are setup for MNIST dataset.

pantry.tutorials.autoencoder.**convolutional_autoencoder**(*dataset=None*, *verbose=1*)

This function is a demo example of a deep convolutional autoencoder. This is an example code. You should study this code rather than merely run it. This is also an example for using the deconvolutional layer or the transposed fractional stride convolutional layers.

Parameters

- **dataset** – Supply a dataset.
- **verbose** – Similar to the rest of the dataset.

pantry.tutorials.autoencoder.**shallow_autoencoder**(*dataset=None*, *verbose=1*)

This function is a demo example of a sparse shallow autoencoder. This is an example code. You should study this code rather than merely run it.

Parameters

- **dataset** – Supply a dataset.
- **verbose** – Similar to the rest of the dataset.

Convolutional Neural Network.

By virture of being here, it is assumed that you have gone through the *Quick Start*.

Building a convolutional neural network is just as similar as an MLNN. The convolutional-pooling layer or convpool layer could be added using the following statement:

```
net.add_layer ( type = "conv_pool",
                origin = "input",
                id = "conv_pool_1",
                num_neurons = 40,
                filter_size = (5,5),
                pool_size = (2,2),
                activation = ('maxout', 'maxout', 2),
                batch_norm = True,
                regularize = True,
                verbose = verbose
            )
```

Here the layer has 40 filters each 5X5 followed by batch normalization followed by a maxpooling of 2X2 all with stride 1. The activation used is maxout with maxout by 2. A simpler relu layer could be added thus,

```
net.add_layer ( type = "conv_pool",
            origin = "input",
            id = "conv_pool_1",
            num_neurons = 40,
            filter_size = (5,5),
            pool_size = (2,2),
            activation = 'relu',
            verbose = verbose
            )
```

Refer to the APIs for more details on the convpool layer. It is often useful to visualize the filters learnt in a CNN, so we introduce the visualizer module here along with the CNN tutorial. The visualizer can be setup using the `add_module` method of `net` object.

```
net.add_module ( type = 'visualizer',
             params = visualizer_params,
             verbose = verbose
            )
```

where the `visualizer_params` is a dictionary of the following format.

```
visualizer_params = {
            "root"       : 'lenet5',
            "frequency"  : 1,
            "sample_size": 144,
            "rgb_filters": True,
            "debug_functions" : False,
            "debug_layers": False,
            "id"         : 'main'
                }
```

`root` is the location where the visualizations are saved, `frequency` is the number of epochs for which visualizations are saved down, `sample_size` number of images are saved each time. `rgb_filters` make the filters save in color. Along with the activities of each layer for the exact same images as the data itself, the filters of neural network are also saved down. For more options of parameters on visualizer refer to the visualizer documentation .

The full code for this tutorial with additional commentary can be found in the file `pantry.tutorials.lenet.py`. This tutorial runs a CNN for the lenet dataset. If you have toolbox cloned or downloaded or just the tutorials downloaded, Run the code using,

Notes

This code contains three methods.

1. A modern reincarnation of LeNet5 for MNIST.
2. **The same Lenet with batchnorms** 2.a. Batchnorm before activations. 2.b. Batchnorm after activations.

All these methods are setup for MNIST dataset.

Todo

Add detailed comments.

`pantry.tutorials.lenet.`**`lenet5`**(*dataset=None*, *verbose=1*)

This function is a demo example of lenet5 from the infamous paper by Yann LeCun. This is an example code. You should study this code rather than merely run it.

> **Warning:** This is not the exact implementation but a modern re-incarnation.

Parameters

- **`dataset`** – Supply a dataset.
- **`verbose`** – Similar to the rest of the dataset.

`pantry.tutorials.lenet.`**`lenet_maxout_batchnorm_after_activation`**(*dataset=None*, *verbose=1*)

This is a version with nesterov momentum and rmsprop instead of the typical sgd. This also has maxout activations for convolutional layers, dropouts on the last convolutional layer and the other dropout layers and this also applies batch norm to all the layers. The difference though is that we use the `batch_norm` layer to apply batch norm that applies batch norm after the activation fo the previous layer. So we just spice things up and add a bit of steroids to *`lenet5()`*. This also introduces a visualizer module usage.

Parameters

- **`dataset`** – Supply a dataset.
- **`verbose`** – Similar to the rest of the dataset.

`pantry.tutorials.lenet.`**`lenet_maxout_batchnorm_before_activation`**(*dataset=None*, *verbose=1*)

This is a version with nesterov momentum and rmsprop instead of the typical sgd. This also has maxout activations for convolutional layers, dropouts on the last convolutional layer and the other dropout layers and this also applies batch norm to all the layers. The batch norm is applied by using the `batch_norm = True` parameters in all layers. This batch norm is applied before activation as is used in the original version of the paper. So we just spice things up and add a bit of steroids to *`lenet5()`*. This also introduces a visualizer module usage.

Parameters

- **`dataset`** – Supply a dataset.
- **`verbose`** – Similar to the rest of the dataset.

Generative Adversarial Networks.

By virture of being here, it is assumed that you have gone through the *Quick Start*.

Todo

Code is done, but text needs to be written in. This code/tutorial will also explain how the network class is setup because to implement a GAN, we need to inherit the network class out and re-write some of the methods.

The full code for this tutorial with additional commentary can be found in the file `pantry.tutorials.gan.py`. If you have toolbox cloned or downloaded or just the tutorials downloaded, Run the code as, Referenced from

Goodfellow, Ian, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. "Generative adversarial nets." In Advances in Neural Information

Processing Systems, pp. 2672-2680. 2014.

Notes

This file contains several GAN implementations:

1. Shallow GAN setup for MNIST
2. Shallow Wasserstein GAN setup for MNIST*
3. Deep GAN (Ian Goodfellow's original implementation) setup for MNIST
4. DCGAN (Chintala et al.) setup for CIFAR 10
5. LS - DCGAN setup for CIFAR 10

Todos:

- Convert the DCGANs for CELEBA.
- WGAN doesn't work properly because of clipping.
- Check that DCGANs strides are properly setup.

`pantry.tutorials.gan.`**`deep_deconvolutional_gan`**(*dataset*, *regularize=True*, *batch_norm=True*, *dropout_rate=0.5*, *verbose=1*)

This function is a demo example of a generative adversarial network. This is an example code. You should study this code rather than merely run it. This method uses a few deconvolutional layers. This method is setup to produce images of size 32X32.

Parameters

- **`dataset`** – Supply a dataset.
- **`regularize`** – `True` (default) supplied to layer arguments
- **`batch_norm`** – `True` (default) supplied to layer arguments
- **`dropout_rate`** – `None` (default) supplied to layer arguments
- **`verbose`** – Similar to the rest of the dataset.

Returns A Network object.

Return type net

Notes

This method is setup for Cifar 10.

pantry.tutorials.gan.**deep_deconvolutional_lsgan**(*dataset*, *regularize=True*, *batch_norm=True*, *dropout_rate=0.5*, *verbose=1*)

This function is a demo example of a generative adversarial network. This is an example code. You should study this code rather than merely run it. This method uses a few deconvolutional layers as was used in the DCGAN paper. This method is setup to produce images of size 32X32.

Parameters

- **dataset** – Supply a dataset.
- **regularize** – `True` (default) supplied to layer arguments
- **batch_norm** – `True` (default) supplied to layer arguments
- **dropout_rate** – `None` (default) supplied to layer arguments
- **verbose** – Similar to the rest of the dataset.

Returns A Network object.

Return type net

Notes

This method is setupfor SVHN / CIFAR10. This is an implementation of th least squares GAN with a = 0, b = 1 and c= 1 (equation 9) [1] Least Squares Generative Adversarial Networks, Xudong Mao, Qing Li, Haoran Xie, Raymond Y.K. Lau, Zhen Wang

pantry.tutorials.gan.**deep_gan_mnist**(*dataset*, *verbose=1*)

This function is a demo example of a generative adversarial network. This is an example code. You should study this code rather than merely run it.

Parameters

- **dataset** – Supply a dataset.
- **verbose** – Similar to the rest of the dataset.

Returns A Network object.

Return type net

Notes

This network here mimics Ian Goodfellow's original code and implementation for MNIST adapted from his source code: https://github.com/goodfeli/adversarial/blob/master/mnist.yaml .It might not be a perfect replicaiton, but I tried as best as I could.

This method is setup for MNIST

pantry.tutorials.gan.**shallow_gan_mnist**(*dataset=None*, *verbose=1*)

This function is a demo example of a generative adversarial network. This is an example code. You should study this code rather than merely run it.

Parameters

- **dataset** – Supply a dataset.
- **verbose** – Similar to the rest of the dataset.

Notes

This method is setup for MNIST.

pantry.tutorials.gan.**shallow_wgan_mnist**(*dataset=None*, *verbose=1*)

This function is a demo example of a Wasserstein generative adversarial network. This is an example code. You should study this code rather than merely run it.

Parameters

- **dataset** – Supply a dataset.
- **verbose** – Similar to the rest of the dataset.

Notes

This method is setup for MNIST. Everything in this code is the same as the shallow GAN class except for the loss functions.

Todo

This is not verified. There is some trouble in weight clipping.

Batch Normalization.

Batch normalization has become one important operation in faster and stable learning of neural networks. In batch norm we do the following:

$$x = (\frac{x - \mu_b}{\sigma_b})\gamma + \beta$$

The x is the input (and the output) of this operation, μ_b and σ_b are the mean and the variance of the minibatch of x supplied. γ and β are learnt using back propagation. This will also store a running mean and a running variance, which is used during inference time.

By default batch normalization can be performed on convolution and dot product layers using the argument `batch_norm = True` supplied to the `yann.network.add_layer` method. This will apply the batch normalization before the activation and after the core layer operation.

While this is the technique that was described in the original batch normalization paper[1]. Some modern networks such as the Residual network [2],[3] use a re-orderd version of layer operations that require the batch norm to be applied post-activation. This is particularly used when using ReLU or Maxout networks[4][5]. Therefore we also provide a layer type `batch_norm`, that could create a layer that simply does batch normalization on the input supplied. These layers could be used to create a post-activation batch normalization.

This tutorial demonstrates the use of both these techniques using the same architecutre of networks used in the *Convolutional Neural Network.* tutorial. The codes for these can be found in the following module methods in `pantry.tutorials`.

References

Notes

This code contains three methods.

1. A modern reincarnation of LeNet5 for MNIST.
2. **The same Lenet with batchnorms** 2.a. Batchnorm before activations. 2.b. Batchnorm after activations.

All these methods are setup for MNIST dataset.

Todo

Add detailed comments.

`pantry.tutorials.lenet.`**`lenet5`**(*dataset=None*, *verbose=1*)

This function is a demo example of lenet5 from the infamous paper by Yann LeCun. This is an example code. You should study this code rather than merely run it.

> **Warning:** This is not the exact implementation but a modern re-incarnation.

Parameters

- **`dataset`** – Supply a dataset.
- **`verbose`** – Similar to the rest of the dataset.

`pantry.tutorials.lenet.`**`lenet_maxout_batchnorm_after_activation`**(*dataset=None*, *verbose=1*)

This is a version with nesterov momentum and rmsprop instead of the typical sgd. This also has maxout activations for convolutional layers, dropouts on the last convolutional layer and the other dropout layers and this also applies batch norm to all the layers. The difference though is that we use the `batch_norm` layer to apply batch norm that applies batch norm after the activation fo the previous layer. So we just spice things up and add a bit of steroids to *`lenet5()`*. This also introduces a visualizer module usage.

Parameters

- **`dataset`** – Supply a dataset.
- **`verbose`** – Similar to the rest of the dataset.

`pantry.tutorials.lenet.`**`lenet_maxout_batchnorm_before_activation`**(*dataset=None*, *verbose=1*)

This is a version with nesterov momentum and rmsprop instead of the typical sgd. This also has maxout activations for convolutional layers, dropouts on the last convolutional layer and the other dropout layers and this also applies batch norm to all the layers. The batch norm is applied by using the `batch_norm = True` parameters in all layers. This batch norm is applied before activation as is used in the original version of the paper. So we just spice things up and add a bit of steroids to *`lenet5()`*. This also introduces a visualizer module usage.

Parameters

- **`dataset`** – Supply a dataset.
- **`verbose`** – Similar to the rest of the dataset.

Cooking a matlab dataset for Yann.

By virture of being here, it is assumed that you have gone through the *Quick Start*.

This tutorial will help you convert a dataset from matlab workspace to yann. To begin let us acquire Google's Street View House Numbers dataset in Matlab [1]. Download from the url three .mat files: test_32x32.mat, train_32x32.mat and extra_32x32.mat. Once downloaded we need to divide this mat dump of data into training, testing and validation minibatches appropriately as used by yann. This can be accomplished by the steps outlined in the code `yann\pantry\matlab\make_svhn.m`. This will create data with 500 samples per mini batch with 56 training batches, 42 testing batches and 28 validation batches.

Once the mat files are setup appropriately, they are ready for yann to load and convert them into yann data. In case of data that is not form svhn, you can open one of the 'batch' files in matlab to understand how the data is spread. Typically, the `x` variable is vectorized images, in this case 500X3072 (500 images per batch, 32*32*3 pixels per image). `y` is an integer vector labels going from 0-10 in this case.

References

To convert the code into yann, we can use the `setup_dataset` module at `yann.utils.dataset.py` file. Simply call the initializer as,

```
dataset = setup_dataset(dataset_init_args = data_params,
                        save_directory = save_directory,
                        preprocess_init_args = preprocess_params,
                        verbose = 3 )
```

where, `data_params` contains information about the dataset thusly,

```
data_params = {
                "source"              : 'matlab',
                # "name"                : 'yann_svhn', # some name.
                "location"                  : location,     # some location to load␣
↪from.
                "height"              : 32,
                "width"               : 32,
                "channels"            : 3,
                "batches2test"        : 42,
                "batches2train"       : 56,
                "batches2validate"    : 28,
                "mini_batch_size"     : 500  }
```

and the `preprocess_params` contains information on how to process the images thusly,

```
preprocess_params = {
                        "normalize"     : True,
                        "ZCA"           : False,
                        "grayscale"     : False,
                        "zero_mean"     : False,
                    }
```

`save_directory` is simply a location to save the yann dataset. Customarialy, it is `save_directory = '_datasets'`

The full code for this tutorial with additional commentary can be found in the file `pantry.tutorials.mat2yann.py`.

If you have toolbox cloned or downloaded or just the tutorials downloaded, Run the code using,

pantry.tutorials.mat2yann.**cook_svhn_normalized**(*location*, *verbose=1*, ***kwargs*)

This method demonstrates how to cook a dataset for yann from matlab. Refer to the pantry/matlab/setup_svhn.m file first to setup the dataset and make it ready for use with yann.

Parameters

- **location** – provide the location where the dataset is created and stored. Refer to prepare_svhn.m file to understand how to prepare a dataset.
- **save_directory** – which directory to save the cooked dataset onto.
- **dataset_parms** – default is the dictionary. Refer to setup_dataset
- **preprocess_params** – default is the dictionary. Refer to setup_dataset.

Notes

By default, this will create a dataset that is not mean-subtracted.

class yann.utils.dataset.**setup_dataset**(*dataset_init_args*, *save_directory='_datasets'*, *verbose=1*, ***kwargs*)

The setup_dataset class is used to create and assemble datasets that are friendly to the Yann toolbox.

Todo

images option for the source. skdata pascal isn't working imagenet dataset and coco needs to be setup.

Parameters

- **dataset_init_args** – is a dictonary of the form:

```
data_init_args = {

    "source" : <where to get the dataset from>
                'pkl' : A theano tutorial style 'pkl' file.
                'skdata' : Download and setup from skdata
                'matlab' : Data is created and is being used from␣
↪Matlab
                'images-only' : Data is created from a directory␣
↪of images. This
                          will be an unsupervised dataset with no␣
↪labels.
    "name" : necessary only for skdata
              supports
                * ``'mnist'``
                * ``'mnist_noise1'``
                * ``'mnist_noise2'``
                * ``'mnist_noise3'``
                * ``'mnist_noise4'``
                * ``'mnist_noise5'``
                * ``'mnist_noise6'``
                * ``'mnist_bg_images'``
                * ``'mnist_bg_rand'``
                * ``'mnist_rotated'``
                * ``'mnist_rotated_bg'``.
                * ``'cifar10'``
                * ``'caltech101'``
```

```
                * ``'caltech256'``

        Refer to original paper by Hugo Larochelle [1] for these␣
↪dataset details.

    "location"                  : necessary for 'pkl' and 'matlab'␣
↪and
                                      'images-only'
    "mini_batch_size"           : 500, # some batch size
    "mini_batches_per_batch"    : (100, 20, 20), # trianing,␣
↪testing, validation
    "batches2train"             : 1, # number of files will be␣
↪created.
    "batches2test"              : 1,
    "batches2validate"          : 1,
    "height"                    : 28, # After pre-processing
    "width"                     : 28,
    "channels"                  : 1 , # color (3) or grayscale (1)␣
↪...

        }
```

- **preprocess_init_args** – provide preprocessing arguments. This is a dictionary:

```
args =  {
    "normalize" : <bool> True for normalize across batches
    "GCN"       : True for global contrast normalization
    "ZCA"       : True, kind of like a PCA representation (not␣
↪fully tested)
    "grayscale" : Convert the image to grayscale
        }
```

- **save_directory** – <string> a location where the dataset is going to be saved.

Notes

Yann toolbox takes datasets in a `.pkl` format. The dataset requires a directory structure such as the following:

```
location/_dataset_XXXXX
|_ data_params.pkl
|_ train
    |_ batch_0.pkl
    |_ batch_1.pkl
    .
    .
    .
|_ valid
    |_ batch_0.pkl
    |_ batch_1.pkl
    .
    .
    .
|_ test
    |_ batch_0.pkl
    |_ batch_1.pkl
    .
```

```
.
.
```

The location id (`XXXXX`) is generated by this class file. The five digits that are produced is the unique id of the dataset.

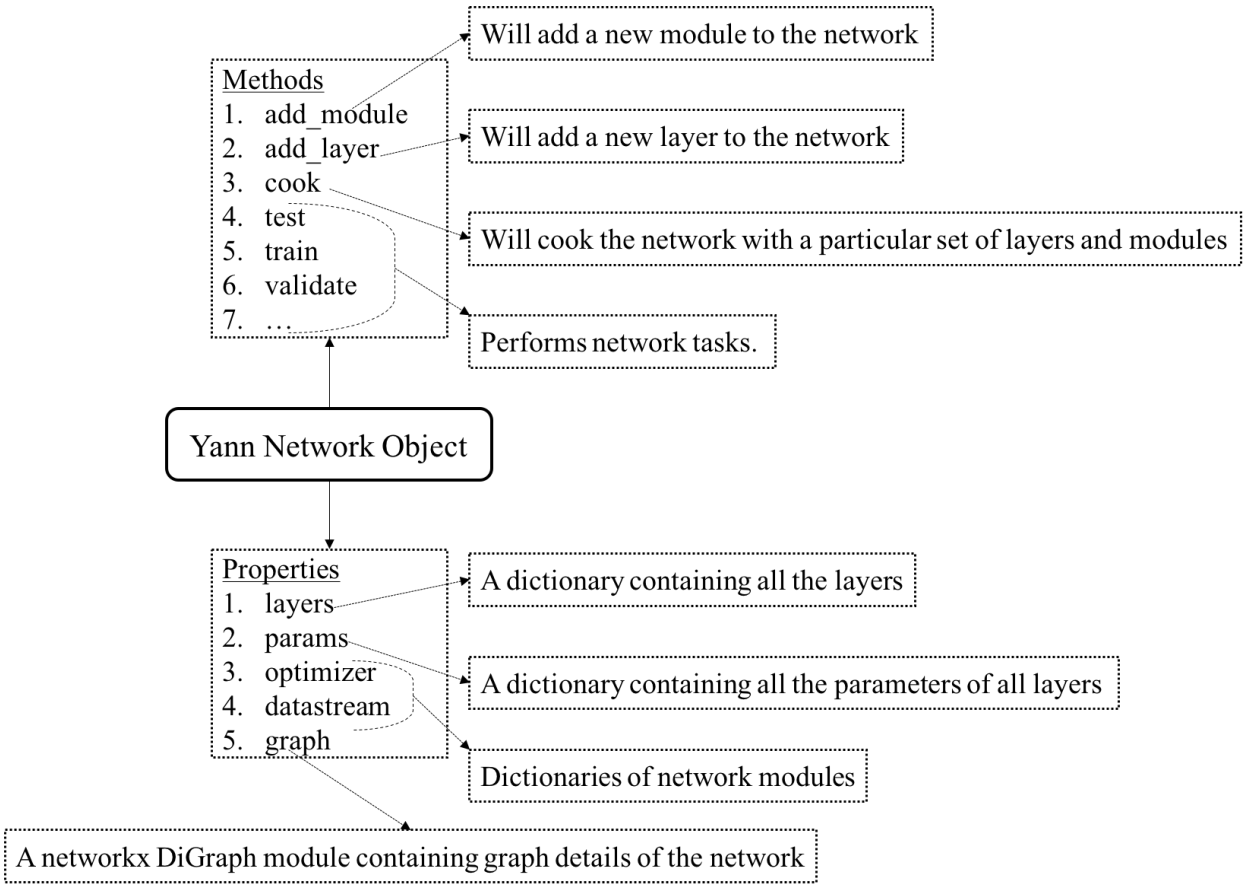
The file `data_params.pkl` contains one variable `dataset_args` used by datastream.

Todo

Do tutorials for the following:

- Loading pre-trained VGG-19 net
- AlexNet
- GoogleNet
- ResNet

Structure of the Yann network



The core of the yann toolbox and its operations are built around the *yann.network.network* class, which is present in the file `yann/network.py`. The above figure shows the organization of the *yann.network.*

network class. The `add_xxxx()` methods add either a layer or module as nomenclature states. The network class can hold various layers and modules in various connections and architecture that are added using the `add_` methods.

Verbose

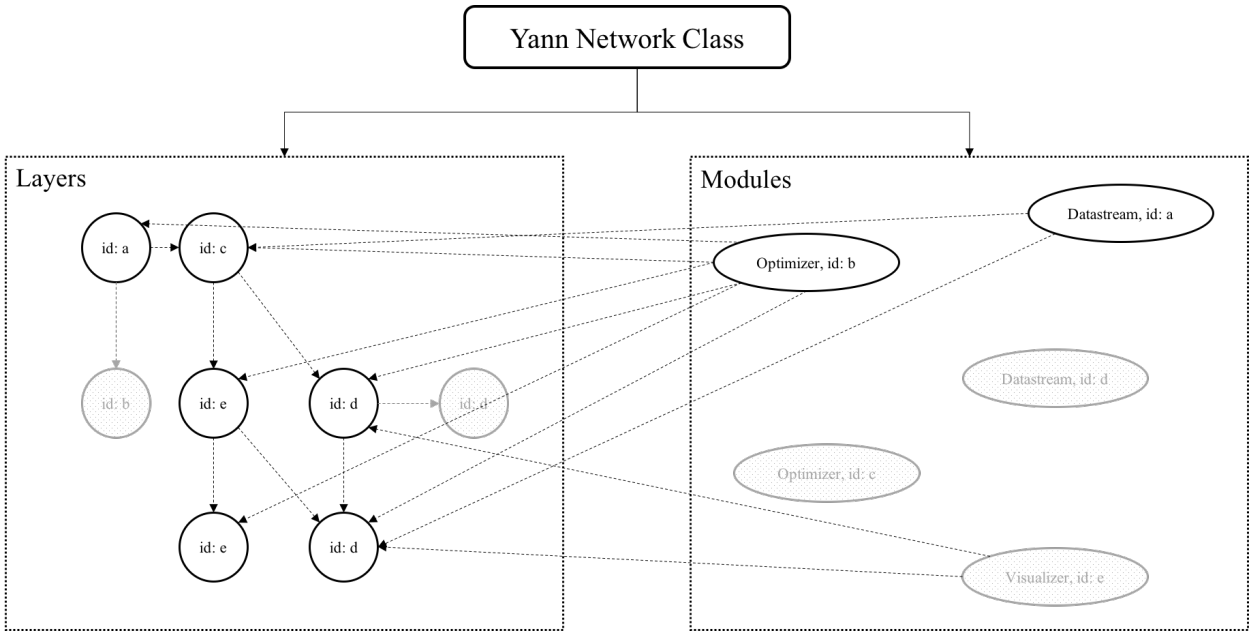
Throughout the toolbox, various methods take an argument called `verbose` as input. `verbose` is by default always `2`. `verbose = 1` implies a silent run and therefore the code doesn't print anything unless absolutely needed. `verbose=2` prints quite the standard amount of information and `verbose==3`, which is friendly when being used for debugging prints annoyingly too much information.

Initializing a network class

A `network` pbject can quite simply be initialized by calling

```
from yann.network import network
net = network()
```

Each layer takes in as argument While prepping the network for learning, we can (or may) need only certain modules and layers. The process of preparing the network by selecting and building the training, testing and validation parts of network is called cooking.



The above figure shows a cooked network. The objects that are in gray and are shaded are uncooked parts of the network. Once cooked, the network is ready for training and testing all by using other methods within the network. The network class also has several properties such as layers, which is a dictionary of the layers that are added to it and params, which is a dictionary of all the parameters. All layers and modules contain a property called *id* through which they are referred.

CHAPTER 2

Quick Start

The easiest way to get going with Yann is to follow this quick start guide. If you are not satisfied and want a more detailed introduction to the toolbox, you may refer to the *Tutorials* and the *Structure of the Yann network*. This tutorial was also presented in CSE591 at ASU and the video of the presentation is available. A more detailed Jupyter Notebook version of this tutorial is available here.

To install in a quick fashion without much dependencies run the follwing command:

```
pip install git+git://github.com/ragavvenkatesan/yann.git
```

If there was an error with installing `skdata`, you might want to install `numpy` and `scipy` independently first and then run the above command. Note that this installer, does not enable a lot of options of the toolbox for which you need to go through the complete install described at the *Installation Guide* page.

Verify that the installation of theano is indeed version 0.9 or greater by doing the following in a python shell

```
import theano
theano.__version__
```

If the version was not 0.9, you can install 0.9 by doing the following:

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

The start and the end of Yann toolbox is the `network` module. The *`yann.network`*. `network` object is where all the magic happens. Start by importing `network` and creating a `network` object in a python shell.

```
from yann.network import network
net = network()
```

Voila! We have thus created a new network. The network doesn't have any layers or modules in it. It be seen verified by probing into `net.layers` property of the `net` object.

```
net.layers
```

This will produce an output which is essentially an empty dictionary `{}`. Let's add some layers! The toolbox comes with a port to skdata the MNIST dataset of handwritten characters can be built using this port.

To cook a mnist dataset for yann run the following code:

```
from yann.special.datasets import cook_mnist
cook_mnist()
```

Running this code will print a statement to the following effect `>>Dataset xxxxx is created.` The five digits marked `xxxxx` in the statement is the codeword for the dataset. The actual dataset is located now at `_datasets/_dataset_xxxxx/` from the directory from where this code was called. Mnist dataset is created and stored at this dataset in a format that is configured for yann to work with. Refer to the *Tutorials* on how to convert your own dataset for yann.

The first layer that we need to add to our network now is an `input` layer. Every ‘‘input‘‘layer requries a dataset to be associated with it. Let us create this layer.

```
dataset_params  = { "dataset": "_datasets/_dataset_xxxxx", "n_classes" : 10 }
net.add_layer(type = "input", dataset_init_args = dataset_params)
```

This piece of code creates and adds a new `datastream` module to the `net` and wires up the newly added `input` layer with this `datastream`. Confirm this by checking `net.datastream`. Let us now build a `classifier` layer. The default classifier that yann is setup with is the logistic regression classifier. Refer to *Toolbox Documentation* or *Tutorials* for other types of layers. Let us create a this `classifier` layer for now.

```
net.add_layer(type = "classifier" , num_classes = 10)
net.add_layer(type = "objective")
```

The layer `objective` creates the loss function from the classifier that can be used as a learning metric. It also provides a scope for other modules such as the `optimizer` module. Refer *Structure of the Yann network* and *Toolbox Documentation* for more details on modules. Now that our network is created and constructed we can see that the `net` objects have `layers` populated.

```
net.layers
>>{'1': <yann.network.layers.classifier_layer object at 0x7eff9a7d0050>, '0':
  <yann.network.layers.input_layer object at 0x7effa410d6d0>, '2':
  <yann.network.layers.objective_layer object at 0x7eff9a71b210>}
```

The keys of the dictionary such as `'1'`, `'0'` and `'2'` are the `id` of the layer. We could have created a layer with a custom id by supplying an `id` argument to the `add_layer` method. To get a better idea of how the network looks like, you can use the `pretty_print` mehtod in yann.

```
net.pretty_print()
```

Now our network is finally ready to be trained. Before training, we need to build an `optimizer` and other tools, but for now let us use the default ones. Once all of this is done, yann requires that the network be ‘cooked’. For more details on cooking refer *Structure of the Yann network*. For now let us imagine that cooking a network will finalize the wiring, architecture, cache and prepare the first batch of data, prepare the modules and in general prepare the network for training using back propagation.

```
net.cook()
```

Cooking would take a few seconds and might print what it is doing along the way. Once cooked, we may notice for instance that the network has a `optimizer` module.

```
net.optimizer
>>{'main': <yann.network.modules.optimizer object at 0x7eff9a7c1b10>}
```

To train the model that we have just cooked, we can use the `train` function that becomes available to us once the network is cooked.

```
net.train()
```

This will print a progress for each epoch and will show validation accuracy after each epoch on a validation set that is independent from the training set. By default the training might run for 40 epochs: 20 on a higher learning rate and 20 more on a fine tuning learning rate.

Every layer also has an `layer.output` object. The `output` can be probed by using the `layer_activity` method as long as it is directly or in-directly associated with a `datastream` module through an `input` layer and the network was cooked. Let us observe the activity of the input layer for trial. Once trained we can observe this output. The layer activity will just be a `numpy` array of numbers, so let us print its shape instead.

```
net.layer_activity(id = '0').shape
net.layers['0'].output_shape
```

The second line of code will verify the output we produced in the first line. An interesting layer output is the output of the `objective` layer, which will give us the current negative log likelihood of the network, the one that we are trying to minimize.

```
net.layer_activity(id = '2')
>>array(0.3926551938056946, dtype=float32)
```

Once we are done training, we can run the network feedforward on the testing set to produce a generalization performance result.

```
net.test()
```

Congratualations, you now know how to use the yann toolbox successfully. A full-fledge code of the logistic regression that we implemented here can be found here . That piece of code also has in-commentary that discusses briefly other options that could be supplied to some of the function calls we made here that explain the processes better.

Hope you liked this quick start guide to the Yann toolbox and have fun!

Toolbox Documentation

Yann toolbox is divided into different parts and written in many files. Below is an index of all files and the API documentations.

`network` - The network module

`yann.network.py` contains the definition for the base `network` class. It is pretty much the most accessible part of this toolbox and forms the structure of the toolbox itself. Any experiment using this toolbox will have to begin and end with using the `network` class:

class `yann.network.`**`network`**(*verbose=2*, ***kwargs*)

Todo:

Class definition for the class `network`:

All network properties, network variables and functionalities are initialized using this class and are contained by the network object. The `network.__init__` method initializes the network class. The `network.__init____` function has many purposes depending on the arguments supplied.

Provide any or all of the following arguments. Appropriate errors will be thrown if the parameters are not supplied correctly.

Todo

•posteriors in a classifier layers is not really a probability. Need to fix this.

Parameters

- **verbose** – Similar to any 3-level verbose in the toolbox.
- **type** – option takes only 'classifier' for now. Will add 'encoders' and others later
- **borrow** – Check `theano's` borrow. Default is `True`.

Returns network object with parameters setup.

Return type `yann.network.network`

add_layer(*type*, *verbose=2*, ***kwargs*)

Todo

Need to add the following: * Inception Layer. * LSTM layer. * ...

Parameters

- **type** – <string> options include 'input' or 'data' - which indicates an input layer. 'conv_pool' or 'convolution' - indicates a convolutional - pooling layer 'deconv' or 'deconvolution' - indicates a fractional stride convoltuion layer 'dot_product' or 'hidden' or 'mlp' or 'fully_connected' - indicates a hidden fully connected layer 'classifier' or 'softmax' or 'output' or 'label' - indicates a classifier layer 'objective' or 'loss' or 'energy' - a layer that creates a loss function 'merge' or 'join' - a layer that merges two layers. 'flatten' - a layer that produces a flattened output of a block data. 'random' - a layer that produces random numbers. 'rotate' - a layer that rotate the input images. 'tensor' - a layer that converts the input tensor as an input layer. From now on everything is optional args..
- **id** – <string> how to identify the layer by. Default is just layer number that starts with `0`.
- **origin** – `id` will use the output of that layer as input to the new layer. Default is the last layer created. This variable for `input` type of layers is not a layer, but a datastream id. For `merge` layer, this is a tuple of two layer ids.
- **verbose** – similar to the rest of the toolbox.
- **mean_subtract** – if `True` we will subtract the mean from each image, else not.
- **num_neurons** – number of neurons in the layer
- **dataset** – <string> Location to the dataset. used when layer `type` is `input`.
- **activation** – String, takes options that are listed in `activations` Needed for layers that use activations. Some activations also take support parameters, for instance `maxout` takes maxout type and size, `softmax` takes an option temperature. Refer to the module `activations` to know more.
- **stride** – tuple `(int , int)`. Used as convolution stride. Default `(1,1)`
- **batch_norm** – If provided will be used, default is `False`.

- **border_mode** – Refer to `border_mode` variable in `yann.core.conv`, module `conv`
- **pool_size** – Subsample size, default is `(1,1)`.
- **pool_type** – Refer to `pool` for details. {'max', 'sum', 'mean', 'max_same_size'}
- **learnable** – Default is `True`, if `True` we backprop on that layer. If `False` Layer is obstinate.
- **shape** – tuple of shape to unflatten to (height, width, channels) in case layer was an unflatten layer
- **input_params** – Supply params or initializations from a pre-trained system.
- **dropout_rate** – If you want to dropout this layer's output provide the output.
- **regularize** – `True` is you want to apply regularization, `False` if not.
- **num_classes** – `int` number of classes to classify.
- **objective** – objective provided by classifier `nll`-negative log likelihood, `cce`-categorical cross entropy, `bce`-binary cross entropy, `hinge`-hinge loss . For classifier layer.
- **dataset_init_args** – same as for the dataset module. In fact this argument is needed only when dataset module is not setup.
- **datastream_id** – When using input layer or during objective layer, use this to identify which datastream to take data from.
- **regularizer** – Default is `(0.001, 0.001)` coeffients for L1, L2 regulaizer coefficients.
- **error** – `merge` layers take an option called `'error'` which can be None or others which are methods in `yann.core.errors`.
- **angle** – Takes value between [0,1] to capture the angle between [0,180] degrees Default is None. If None is specified, random number is generated from a uniform distriibution between 0 and 1.
- **layer_type** – If `value` supply, else it is default `'discriminator'`. For other layers, if the layer class takes an argument `type`, supply that argument here as `layer_type`. `merge` layer for instance will use this arugment as its `type` argument.

add_module(*type*, *params=None*, *verbose=2*)

Use this function to add a module to the net.

Parameters

- **type** – which module to add. Options are `'resultor'`, `'visualizer'`, `'optimizer'` `'datastream'`
- **params** – If the `type` was `'resultor'` params is a dictionary of the form:

```
params    =    {
        "root"     : "<root directory to save stuff inside>"
        "results"  : "<results_file_name>.txt",
        "errors"   : "<error_file_name>.txt",
        "costs"    : "<cost_file_name>.txt",
        "confusion": "<confusion_file_name>.txt",
        "network"  : "<network_save_file_name>.pkl"
        "id"       : id of the resultor
                }
```

While the filenames are optional, `root` must be provided. If a particular file is not provided, that value will not be saved. This value is supplied to setup the resultor module of :mod: *network*.

If the `type` was `'visualizer'` params is a dictionary of the form:

```
parmas = {
        "root"        : location to save the visualizations
        "frequency"   : <integer>, after how many epochs do you need to
                        visualize. Default value is 1import os

        "sample_size" : <integer, prefer squares>, simply save down random
                        images from the datasets also saves down activations
                        for the same images also. Default value is 16
        "rgb_filters" : <bool> flag. if True 3D-RGB CNN filters are rendered.
                        Default value is False
        "id"          : id of the visualizer
                }
```

If the `type` was `'optimizer'` params is a dictionary of the form:

```
params =  {
        "momentum_type"       : <option> takes 'false' <no momentum>, 'polyak'
                                and 'nesterov'. Default value is 'polyak'
        "momentum_params"   : (<value in [0,1]>, <value in [0,1]>, <int>),
                                (momentum coeffient at start, at end, at what
                                epoch to end momentum increase). Default is
                                the tuple (0.5, 0.95,50)
        "learning_rate"   : (initial_learning_rate, fine_tuning_learning_rate,
                                annealing_decay_rate). Default is the tuple
                                (0.1,0.001,0.005)
        "regularization"    : (l1_coeff, l2_coeff). Default is (0.001, 0.001)
        "optimizer_type": <option>, takes 'sgd', 'adagrad', 'rmsprop', 'adam'.
                                Default is 'rmsprop'
        "objective_function": <option>,  takes
                                'nll'-negative log likelihood,
                                'cce'-categorical cross entropy,
                                'bce'-binary cross entropy.
                                Default is 'nll'
        "id"                : id of the optimizer
            }
```

If the `type was ``'datastream'` params is a dictionary of the form:

```
params = {
            "dataset":  <location>
            "svm"    :  False or True
                ``svm`` if ``True``, a one-hot label set will␣
↪also be setup.
            "n_classes": <int>
                ``n_classes`` if ``svm`` is ``True``, we need to␣
↪know how
                many ``n_classes`` are present.
            "id": id of the datastream
    }
```

- **verbose** – Similar to rest of the toolbox.

cook (*verbose=2*, ***kwargs*)

This function builds the backprop network, and makes the trainer, tester and validator theano functions. The trainer builds the trainers for a particular objective layer and optimizer.

Parameters

- **optimizer** – Supply which optimizer to use. Default is last optimizer created.
- **datastream** – Supply which datastream to use. Default is the last datastream created.
- **visualizer** – Supply a visualizer to cook with. Default is the last visualizer created.
- **classifier_layer** – supply the layer of classifier. Default is the last classifier layer created.
- **objective_layers** – Supply a list of layer ids of layers that has the objective function. Default is last objective layer created if no classifier is provided.
- **objective_weights** – Supply a list of weights to be multiplied by each value of the objective layers. Default is 1.
- **active_layers** – Supply a list of active layers. If this parameter is supplied all `'learnabile'` of all layers will be ignored and only these layers will be trained. By default, all the learnable layers are used.
- **verbose** – Similar to the rest of the toolbox.

deactivate_layer (*id*, *verbose=2*)

This method will remove a layer's parameters from the active_layer dictionary.

Parameters

- **id** – Layer which you want to de activate.
- **verbose** – as usual.

Notes

If the network was cooked, it would have to be re-cooked after deactivation.

get_params (*verbose=2*)

This method returns a dictionary of layer weights and bias in numpy format.

Parameters **verbose** – Blah..

Returns A dictionary of parameters.

Return type OrderedDict

layer_activity(*id*, *index=0*, *verbose=2*)
Use this function to visualize or print out the outputs of each layer. I don't know why this might be useful, but its fun to check this out I guess. This will only work after the dataset is initialized.

Parameters

- **id** – id of the layer that you want to visualize the output for.
- **index** – Which batch of data should I use for producing the outputs. Default is `0`

pretty_print(*verbose=2*)
This method is used to pretty print the network's connections This is going to be deprecated with the use of visualizer module.

print_status(*epoch*, *print_lr=False*, *verbose=2*)
This function prints the cost of the current epoch, learning rate and momentum of the network at the moment. This also calls the resultor to process results.

Todo

This needs to to go to visualizer.

Parameters

- **verbose** – Just as always.
- **epoch** – Which epoch are we at ?

save_params(*epoch=0*, *verbose=2*)
This method will save down a list of network parameters

Parameters

- **verbose** – As usual
- **epoch** – epoch.

test(*show_progress=True*, *verbose=2*)
This function is used for producing the testing accuracy.

Parameters **verbose** – As usual

train(*verbose=2*, ***kwargs*)
Training function of the network. Calling this will begin training.

Parameters

- **epochs** – `(num_epochs for each learning rate... )` to train Default is `(20, 20)`
- **validate_after_epochs** – 1, after how many epochs do you want to validate ?
- **save_after_epochs** – 1, Save network after that many epochs of training.
- **show_progress** – default is `True`, will display a clean progressbar. If `verbose` is `3` or more - False
- **early_terminate** – `True` will allow early termination.
- **learning_rates** – (annealing_rate, learning_rates ...) length must be one more than `epochs` Default is `(0.05, 0.01, 0.001)`

validate(*epoch=0*, *training_accuracy=False*, *show_progress=False*, *verbose=2*)
Method is use to run validation. It will also load the validation dataset.

Parameters

- **verbose** – Just as always
- **show_progress** – Display progressbar ?
- **training_accuracy** – Do you want to print accuracy on the training set as well ?

visualize(*epoch=0*, *verbose=2*)
This method will use the cooked visualizer to save down the visualizations

Parameters **epoch** – supply the epoch number (used to create directories to save

visualize_activities(*epoch=0*, *verbose=2*)
This method will save down all layer activities for the correct epoch.

Parameters

- **epoch** – What epoch is being running now.
- **verbose** – As always.

visualize_filters(*epoch=0*, *verbose=2*)
This method will save down all layer filters for the correct epoch.

Parameters

- **epoch** – What epoch is being running now.
- **verbose** – As always.

`layers` - Contains the definitions of all the types of layers.

The module `yann.layers` contains the definition for the different types of layers that are accessible in `yann`. It contains various layers including:

- `abstract.layer`
- `input.input_layer`
- **`fully_connected.dropout_dot_product_layer` and** `fully_connected.dot_product_layer`
- **`conv_pool.dropout_conv_pool_layer_2d` and** `conv_pool.conv_pool_layer_2d`
- `ouput.classifier_layer`
- `output.objective_layer`
- `merge.merge_layer`
- `flatten.flatten_layer`
- `flatten.unflatten_layer`
- `random.random_layer`
- **`batch_norm.batch_norm_layer_2d` and** `batch_norm.dropout_batch_norm_layer_2d`
- **`batch_norm.batch_norm_layer_1d` and** `batch_norm.dropout_batch_norm_layer_1d`

All these are inherited classes from `layer` class, which is abstract.

Specific layers that can be used are

`abstract` - abstraction class

The file `yann.layers.abstract.py` contains the definition for the abstract layer:

Todo

- LSTM / GRN layers
- An Embed layer that is going to create a new embedding space for two layer's activations to project on to the same space and minimize its distances.

---

class `yann.layers.abstract.`**`layer`**(*id*, *type*, *verbose=2*)

Prototype for what a layer should look like. Every layer should inherit from this. This is a template class do not use this directly, you need to use a specific type of layer which again will be called by `yann.network.network.add_layer`

Parameters

- **`id`** – String
- **`origin`** – String id
- **`type`** – string- `'classifier'`, `'dot-product'`, `'objective'`, `'conv_pool'`, `'input'` .. .

Notes

Use `self.type`, `self.origin`, self.destination`‘‘`, `self.output`, `self.inference` `self.output_shape` for outside calls and purposes.

`get_params`(*borrow=True*, *verbose=2*)

This method returns the parameters of the layer in a numpy ndarray format.

Parameters

- **`borrow`** – Theano borrow, default is True.
- **`verbose`** – As always

Notes

This is a slow method, because we are taking the values out of GPU. Ordinarily, I should have used get_value(borrow = True), but I can't do this because some parameters are theano.tensor.var.TensorVariable which needs to be run through eval.

`print_layer`(*prefix=' '*, *nest=True*, *last=True*, *verbose=2*)

Print information about the layer

Parameters

- **`nest`** – If True will print the tree from here on. If False it will print only this layer.
- **`prefix`** – Is what prefix you want to add to the network print command.

`input` - input layer classes

The file `yann.layers.input.py` contains the definition for the input lyer modules.

class `yann.layers.input.`**`dropout_input_layer`**(*mini_batch_size*, *id*, *x*, *dropout_rate=0.5*, *height=28*, *width=28*, *channels=1*, *mean_subtract=False*, *rng=None*, *verbose=2*)

Creates a new input_layer. The layer doesn't do much except to take the networks' x and reshapes it into images. This is needed because yann dataset module assumes data in vectorized image formats as used in mnist - theano tutorials.

This class also creates a branch between a mean subtracted and non-mean subtracted input. It always assumes as default to use the non-mean subtracted input but if `mean_subtract` flag is provided, it will use the other option.

Parameters

- **`x`** – `theano.tensor` variable with rows are vectorized images. if None, will create a new one.
- **`mini_batch_size`** – Number of images in the data variable.
- **`height`** – Height of each image.
- **`width`** – Width of each image.
- **`channels`** – Number of channels in each image.
- **`mean_subtract`** – Defauly is `False`.
- **`verbose`** – Similar to all of the toolbox.

Notes

Use `input_layer.output` to continue onwards with the network `input_layer.output_shape` will tell you the output size.

class `yann.layers.input.`**`dropout_tensor_layer`**(*id*, *input*, *input_shape*, *rng=None*, *dropout_rate=0.5*, *verbose=2*)

This converts a theano tensor or a shared value into a layer. Simply the value becomes the layer's outptus.

Parameters

- **`input`** – some tensor
- **`input_shape`** – shape of the tensor
- **`dropout_rate`** – default is 0.5, typically.
- **`rng`** – Random number generator
- **`verbose`** – Similar to all of the toolbox.

Notes

Use `input_layer.output` to continue onwards with the network `input_layer.output_shape` will tell you the output size.

class `yann.layers.input.`**`input_layer`**(*mini_batch_size*, *x*, *id=-1*, *height=28*, *width=28*, *channels=1*, *mean_subtract=False*, *verbose=2*)

reshapes it into images. This is needed because yann dataset module assumes data in vectorized image formats as used in mnist - theano tutorials.

This class also creates a branch between a mean subtracted and non-mean subtracted input. It always assumes as default to use the non-mean subtracted input but if `mean_subtract` flag is provided, it will use the other option.

Parameters

- **x** – `theano.tensor` variable with rows are vectorized images.
- **y** – `theano.tensor` variable
- **one_hot_y** – `theano.tensor` variable
- **mini_batch_size** – Number of images in the data variable.
- **height** – Height of each image.
- **width** – Width of each image.
- **id** – Supply a layer id
- **channels** – Number of channels in each image.
- **mean_subtract** – Defauly is `False`.
- **verbose** – Similar to all of the toolbox.

Notes

Use `input_layer.output` to continue onwards with the network `input_layer.output_shape` will tell you the output size. Use `input_layer.x`, `input_layer.y` and `input_layer_one_hot_y` tensors for connections.

class `yann.layers.input.`**tensor_layer**(*id*, *input*, *input_shape*, *verbose=2*)

This converts a theano tensor or a shared value into a layer. Simply the value becomes the layer's outptus.

Parameters

- **input** – some tensor
- **input_shape** – shape of the tensor
- **verbose** – Similar to all of the toolbox.

Notes

Use `input_layer.output` to continue onwards with the network `input_layer.output_shape` will tell you the output size.

`fully_connected` - fully connected layer classes

The file `yann.layers.fully_connected.py` contains the definition for the fc layers.

class `yann.layers.fully_connected.`**dot_product_layer**(*input*, *num_neurons*, *input_shape*, *id*, *rng=None*, *input_params=None*, *borrow=True*, *activation='relu'*, *batch_norm=True*, *verbose=2*)

This class is the typical neural hidden layer and batch normalization layer. It is called by the `add_layer` method in network class.

Parameters

- **input** – An input `theano.tensor` variable. Even `theano.shared` will work as long as they are in the following shape `mini_batch_size, height, width, channels`
- **verbose** – similar to the rest of the toolbox.
- **num_neurons** – number of neurons in the layer
- **input_shape** – `(mini_batch_size, input_size)` theano shared
- **batch_norm** – If provided will be used, default is `False`.
- **rng** – typically `numpy.random`.
- **borrow** – `theano` borrow, typicall `True`.
- **activation** – String, takes options that are listed in `activations` Needed for layers that use activations. Some activations also take support parameters, for instance `maxout` takes maxout type and size, `softmax` takes an option temperature. Refer to the module `activations` to know more.
- **input_params** – Supply params or initializations from a pre-trained system.

Notes

Use `dot_product_layer.output` and `dot_product_layer.output_shape` from this class. `L1` and `L2` are also public and can also can be used for regularization. The class also has in public `w`, `b` and `alpha` which are also a list in `params`, another property of this class.

L2 = None

Ioffe, Sergey, and Christian Szegedy. "Batch normalization – Accelerating deep network training by reducing internal covariate shift." arXiv preprint arXiv:1502.03167 (2015).

class `yann.layers.fully_connected.`**dropout_dot_product_layer**(*input*, *num_neurons*, *input_shape*, *id*, *dropout_rate=0.5*, *rng=None*, *input_params=None*, *borrow=True*, *activation='relu'*, *batch_norm=True*, *verbose=2*)

This class is the typical dropout neural hidden layer and batch normalization layer. Called by the `add_layer` method in network class.

Parameters

- **input** – An input `theano.tensor` variable. Even `theano.shared` will work as long as they are in the following shape `mini_batch_size, height, width, channels`
- **verbose** – similar to the rest of the toolbox.
- **num_neurons** – number of neurons in the layer
- **input_shape** – `(mini_batch_size, input_size)`
- **batch_norm** – If provided will be used, default is `False`.
- **rng** – typically `numpy.random`.
- **borrow** – `theano` borrow, typicall `True`.

- **activation** – String, takes options that are listed in `activations` Needed for layers that use activations. Some activations also take support parameters, for instance `maxout` takes maxout type and size, `softmax` takes an option temperature. Refer to the module `activations` to know more.
- **input_params** – Supply params or initializations from a pre-trained system.
- **dropout_rate** – `0.5` is the default.

Notes

Use `dropout_dot_product_layer.output` and `dropout_dot_product_layer.output_shape` from this class. `L1` and `L2` are also public and can also can be used for regularization. The class also has in public `w`, `b` and `alpha` which are also a list in `params`, another property of this class.

`conv_pool` - conv pool layer classes

The file `yann.layers.conv_pool.py` contains the definition for the conv pool layers.

Todo

- Need to the deconvolutional-unpooling layer.
- Something is still not good about the convolutional batch norm layer.

class `yann.layers.conv_pool.`**conv_pool_layer_2d**(*input*, *nkerns*, *input_shape*, *id*, *filter_shape=(3, 3)*, *poolsize=(2, 2)*, *pooltype='max'*, *batch_norm=False*, *border_mode='valid'*, *stride=(1, 1)*, *rng=None*, *borrow=True*, *activation='relu'*, *input_params=None*, *verbose=2*)

This class is the typical 2D convolutional pooling and batch normalizationlayer. It is called by the `add_layer` method in network class.

Parameters

- **input** – An input `theano.tensor` variable. Even `theano.shared` will work as long as they are in the following shape `mini_batch_size, height, width, channels`
- **verbose** – similar to the rest of the toolbox.
- **nkerns** – number of neurons in the layer
- **input_shape** – `(mini_batch_size, channels, height, width)`
- **filter_shape** – (<int>,<int>)
- **pool_size** – Subsample size, default is `(1,1)`.
- **pool_type** – Refer to `pool` for details. {'max', 'sum', 'mean', 'max_same_size'}
- **batch_norm** – If provided will be used, default is `False`.
- **border_mode** – Refer to `border_mode` variable in `yann.core.conv`, module *conv*
- **stride** – tuple `(int , int)`. Used as convolution stride. Default `(1,1)`

- **rng** – typically `numpy.random`.
- **borrow** – `theano` borrow, typicall `True`.
- **activation** – String, takes options that are listed in `activations` Needed for layers that use activations. Some activations also take support parameters, for instance `maxout` takes maxout type and size, `softmax` takes an option temperature. Refer to the module `activations` to know more.
- **input_params** – Supply params or initializations from a pre-trained system.

Notes

Use `conv_pool_layer_2d.output` and `conv_pool_layer_2d.output_shape` from this class. `L1` and `L2` are also public and can also can be used for regularization. The class also has in public `w`, `b`, `gamam`, `beta`,''running_mean'' and `running_var` which are also a list in `params`, another property of this class.

print_layer(*prefix=' '*, *nest=False*, *last=True*)
Print information about the layer

class `yann.layers.conv_pool.`**deconv_layer_2d**(*input*, *nkerns*, *input_shape*, *id*, *output_shape*, *filter_shape=(3, 3)*, *poolsize=(1, 1)*, *pooltype='max'*, *batch_norm=False*, *border_mode='valid'*, *stride=(1, 1)*, *rng=None*, *borrow=True*, *activation='relu'*, *input_params=None*, *verbose=2*)

This class is the typical 2D convolutional pooling and batch normalizationlayer. It is called by the `add_layer` method in network class.

Parameters

- **input** – An input `theano.tensor` variable. Even `theano.shared` will work as long as they are in the following shape `mini_batch_size, height, width, channels`
- **verbose** – similar to the rest of the toolbox.
- **nkerns** – number of neurons in the layer
- **input_shape** – `(mini_batch_size, channels, height, width)`
- **filter_shape** – (<int>,<int>)
- **pool_size** – Subsample size, default is `(1,1)`. Right now does not take a pooling.
- **pool_type** – Refer to `pool` for details. {'max', 'sum', 'mean', 'max_same_size'}
- **batch_norm** – If provided will be used, default is `False`.
- **border_mode** – Refer to `border_mode` variable in `yann.core.conv`, module *conv*
- **stride** – tuple `(int , int)`. Used as convolution stride. Default `(1,1)`
- **rng** – typically `numpy.random`.
- **borrow** – `theano` borrow, typicall `True`.
- **activation** – String, takes options that are listed in `activations` Needed for layers that use activations. Some activations also take support parameters, for instance `maxout` takes maxout type and size, `softmax` takes an option temperature. Refer to the module `activations` to know more.
- **input_params** – Supply params or initializations from a pre-trained system.

Notes

Use `conv_pool_layer_2d.output` and `conv_pool_layer_2d.output_shape` from this class. `L1` and `L2` are also public and can also can be used for regularization. The class also has in public `w`, `b`, `gamam`, `beta`,''running_mean'' and `running_var` which are also a list in `params`, another property of this class.

print_layer(*prefix=' '*, *nest=False*, *last=True*)
Print information about the layer

class yann.layers.conv_pool.**dropout_conv_pool_layer_2d**(*input*, *nkerns*, *input_shape*, *id*, *dropout_rate=0.5*, *filter_shape=(3, 3)*, *poolsize=(2, 2)*, *pooltype='max'*, *batch_norm=True*, *border_mode='valid'*, *stride=(1, 1)*, *rng=None*, *borrow=True*, *activation='relu'*, *input_params=None*, *verbose=2*)

This class is the typical 2D convolutional pooling and batch normalizationlayer. It is called by the `add_layer` method in network class.

Parameters

- **input** – An input `theano.tensor` variable. Even `theano.shared` will work as long as they are in the following shape `mini_batch_size, height, width, channels`
- **verbose** – similar to the rest of the toolbox.
- **nkerns** – number of neurons in the layer
- **input_shape** – `(mini_batch_size, channels, height, width)`
- **filter_shape** – (<int>,<int>)
- **pool_size** – Subsample size, default is `(1,1)`.
- **pool_type** – Refer to `pool` for details. {'max', 'sum', 'mean', 'max_same_size'}
- **batch_norm** – If provided will be used, default is `False`.
- **border_mode** – Refer to `border_mode` variable in `yann.core.conv`, module `conv`
- **stride** – tuple `(int , int)`. Used as convolution stride. Default `(1,1)`
- **rng** – typically `numpy.random`.
- **borrow** – `theano` borrow, typicall `True`.
- **activation** – String, takes options that are listed in `activations` Needed for layers that use activations. Some activations also take support parameters, for instance `maxout` takes maxout type and size, `softmax` takes an option temperature. Refer to the module `activations` to know more.
- **input_params** – Supply params or initializations from a pre-trained system.

Notes

Use `conv_pool_layer_2d.output` and `conv_pool_layer_2d.output_shape` from this class. `L1` and `L2` are also public and can also can be used for regularization. The class also has in public `w`, `b` and

alpha which are also a list in params, another property of this class.

class yann.layers.conv_pool.**dropout_deconv_layer_2d**(*input*, *nkerns*, *input_shape*, *id*, *output_shape*, *dropout_rate=0.5*, *filter_shape=(3, 3)*, *poolsize=(1, 1)*, *pooltype='max'*, *batch_norm=True*, *border_mode='valid'*, *stride=(1, 1)*, *rng=None*, *borrow=True*, *activation='relu'*, *input_params=None*, *verbose=2*)

This class is the typical 2D deconvolutional and batch normalization layer. It is called by the add_layer method in network class.

Parameters

- **input** – An input theano.tensor variable. Even theano.shared will work as long as they are in the following shape mini_batch_size, height, width, channels
- **verbose** – similar to the rest of the toolbox.
- **nkerns** – number of neurons in the layer
- **input_shape** – (mini_batch_size, channels, height, width)
- **filter_shape** – (<int>,<int>)
- **pool_size** – Subsample size, default is (1,1).
- **pool_type** – Refer to pool for details. {'max', 'sum', 'mean', 'max_same_size'}
- **batch_norm** – If provided will be used, default is False.
- **border_mode** – Refer to border_mode variable in yann.core.conv, module conv
- **stride** – tuple (int , int). Used as convolution stride. Default (1,1)
- **rng** – typically numpy.random.
- **borrow** – theano borrow, typicall True.
- **activation** – String, takes options that are listed in activations Needed for layers that use activations. Some activations also take support parameters, for instance maxout takes maxout type and size, softmax takes an option temperature. Refer to the module activations to know more.
- **input_params** – Supply params or initializations from a pre-trained system.

Notes

Use conv_pool_layer_2d.output and conv_pool_layer_2d.output_shape from this class. L1 and L2 are also public and can also can be used for regularization. The class also has in public w, b and alpha which are also a list in params, another property of this class.

merge - merge layer classes

The file yann.layers.merge.py contains the definition for the merge layers.

class yann.layers.merge.**dropout_merge_layer**(*x*, *input_shape*, *id=-1*, *error='rmse'*, *rng=None*, *dropout_rate=0.5*, *verbose=2*)

This is a dropotu merge layer. This takes two layer inputs and produces an error between them if the `type` argument supplied was `'error'`. It does other things too accordingly.

Parameters

- **x** – a list of inputs (lenght must be two basically)
- **input_shape** – List of the shapes of all inputs.
- **type** – `'error'` creates an error layer. other options are `'sum'` and `'concatenate'`
- **error** – If the type was `'error'`, then this variable is used. options include, `'rmse'`, `'l2'`, `'l1'`,‘’’cross_entropy’‘‘.

class yann.layers.merge.**merge_layer**(*x*, *input_shape*, *id=-1*, *type='error'*, *error='rmse'*, *input_type='layer'*, *verbose=2*)

This is a merge layer. This takes two layer inputs and produces an error between them if the `type` argument supplied was `'error'`. It does other things too accordingly.

Parameters

- **x** – a list of inputs (lenght must be two basically)
- **input_shape** – List of the shapes of all inputs.
- **type** – `'error'` creates an error layer. other options are `'sum'`, `'batch'` and `'concatenate'`
- **error** – If the type was `'error'`, then this variable is used. options include, `'rmse'`, `'l2'`, `'l1'`,‘’’cross_entropy’‘‘.
- **input_type** – If this argument was `'tensor'`, we simply merge the ouptus, if this was not provided or was `'layer'`, this merges the outputs of the two layers.

Notes

`'concatenate'` concatenates the outputs on the channels where as `'batch'` concatenates across the batches. It will increase the batchsize.

loss(*type=None*)

This method will return the cost function of the merge layer. This can be used by the optimizer module for instance to acquire a symbolic loss function that tries to minimize the distance between the two layers.

Parameters

- **y** – symbolic `theano.ivector` variable of labels to calculate loss from.
- **type** – options None - Simple error ‘log’ - log loss

Returns loss value.

Return type theano symbolic variable

output_shape = None

if len(input_shape) == 2 – self.num_neurons = self.output_shape[-1] elif len(input_shape) == 4:

self.num_neurons = self.output_shape[1]

`flatten` - flatten layer classes

The file `yann.layers.flatten.py` contains the definition for the flatten layers.

class `yann.layers.flatten.`**`flatten_layer`**(*input*, *input_shape*, *id=-1*, *verbose=2*)
This is a flatten layer. This takes a square layer and flatten it.

Parameters

- **`input`** – output of some layer.
- **`id`** – id of the layer
- **`verbose`** – as usual

class `yann.layers.flatten.`**`unflatten_layer`**(*input*, *shape*, *input_shape*, *id=-1*, *verbose=2*)
This is an unflatten layer. This takes a flattened input and unflattens it.

Parameters

- **`input`** – output of some layer.
- **`shape`** – shape to unflatten.
- **`id`** – id of the layer
- **`verbose`** – as usual.

`output` - output layer classes

The file `yann.layers.output.py` contains the definition for the conv pool layers.

class `yann.layers.output.`**`classifier_layer`**(*input*, *input_shape*, *id*, *num_classes=10*, *rng=None*, *input_params=None*, *borrow=True*, *activation='softmax'*, *verbose=2*)
This class is the typical classifier layer. It should be called by the `add_layer` method in network class.

Parameters

- **`input`** – An input `theano.tensor` variable. Even `theano.shared` will work as long as they are in the following shape `mini_batch_size, height, width, channels`
- **`verbose`** – similar to the rest of the toolbox.
- **`input_shape`** – `(mini_batch_size, features)`
- **`num_classes`** – number of classes to classify into
- **`filter_shape`** – (<int>,<int>)
- **`batch_norm`** – <bool> (Not active yet. Will be implemented in near future.)
- **`rng`** – typically `numpy.random`.
- **`borrow`** – `theano` borrow, typicall `True`.
- **`rng`** – typically `numpy.random`.
- **`activation`** – String, takes options that are listed in `activations` Needed for layers that use activations. Some activations also take support parameters, for instance `maxout` takes maxout type and size, `softmax` takes an option temperature. Refer to the module `activations` to know more. Default is 'softmax'
- **`input_params`** – Supply params or initializations from a pre-trained system.

Notes

Use `classifier_layer.output` and `classifier_layer.output_shape` from this class. `L1` and `L2` are also public and can also can be used for regularization. The class also has in public `w`, `b` and `alpha` which are also a list in `params`, another property of this class.

errors(*y*)
This function returns a count of wrong predictions.

Parameters **y** – datastreamer's `y` variable, that has the lables.

Returns number of wrong predictions.

Return type theano variable

get_params(*borrow=True*, *verbose=2*)
This method returns the parameters of the layer in a numpy ndarray format.

Parameters

- **borrow** – Theano borrow, default is True.
- **verbose** – As always

Notes

This is a slow method, because we are taking the values out of GPU. Ordinarily, I should have used get_value(borrow = True), but I can't do this because some parameters are theano.tensor.var.TensorVariable which needs to be run through eval.

loss(*y*, *type*)
This method will return the cost function of the classifier layer. This can be used by the optimizer module for instance to acquire a symbolic loss function.

Parameters

- **y** – symbolic `theano.ivector` variable of labels to calculate loss from.
- **type** – options 'nll' - negative log likelihood, 'cce' - categorical cross entropy, 'bce' - binary cross entropy, 'hinge' - max-margin hinge loss.

Returns loss value.

Return type theano symbolic variable

class `yann.layers.output.`**objective_layer**(*id*, *loss*, *labels=None*, *objective='nll'*, *L1=None*, *L2=None*, *l1_coeff=0.001*, *l2_coeff=0.001*, *verbose=2*)
This class is an objective layer. It just has a wrapper for loss function. I need this because I am making objective as a loss layer.

Parameters

- **loss** – `yann.network.layers.classifier_layer.loss()` method, or some thenao variable if other types of objective layers.
- **labels** – `theano.shared` variable of labels.
- **objective** – `'nll'`, `'cce'`, `'nll'` or `''bce''` or `'hinge'` for classifier kayers. `'value'`. Value will just use the value as an objective and minimizes that. depends on what is the classifier layer being used.

  Each have their own options. This is usually a string.

- **L1** – Symbolic weight of the L1 added together
- **L2** – Sumbolic L2 of the weights added together
- **l1_coeff** – Coefficient to weight L1 by.
- **l2_coeff** – Coefficient to weight L2 by.
- **verbose** – Similar to the rest of the toolbox.

Todo

The loss method needs to change in input.

Notes

Use `objective_layer.output` and from this class.

`random` - random layer classes

The file `yann.layers.random.py` contains the definition for the merge layers.

class `yann.layers.random.`**random_layer**(*num_neurons*, *id=-1*, *distribution='binomial'*, *verbose=2*, *options=None*)

This is a random generation layer.

Parameters

- **num_neurons** – List of the shapes of all inputs.
- **distribution** – `'binomial'`, `'uniform'`, `'normal'` `'gaussian'`
- **limits** – tuple for uniform
- **mu** – mean for gaussian
- **sigma** – variance for gaussian
- **p** – if `type` is `'binomial'` supply a `p` variable. Default is 0.5
- **id** – Supply a layer id
- **num_neurons** – Supply the output shape of the layer desired.
- **verbose** – As always

`transform` - transform layers

The file `yann.layers.transform.py` contains the definition for the transformation layers. This code is used to rotate the images given some angles between [0,1].

Obliging License, credit and conditions for Lasagne: Some part of the file was directly reproduced from the Lasagne code base.

Author: Anchit Agarwal

LICENSE

class `yann.layers.transform.`**`dropout_rotate_layer`**(*input*, *input_shape*, *id*, *rng=None*, *dropout_rate=0.5*, *angle=None*, *borrow=True*, *verbose=2*)

This class is the typical dropout neural hidden layer and batch normalization layer. Called by the `add_layer` method in network class.

Parameters

- **`input`** – An input `theano.tensor` variable. Even `theano.shared` will work as long as they are in the following shape `mini_batch_size, channels, height, width`
- **`verbose`** – similar to the rest of the toolbox.
- **`input_shape`** – `(mini_batch_size, input_size)`
- **`angle`** – value from [0,1]
- **`borrow`** – `theano` borrow, typically `True`.
- **`rng`** – typically `numpy.random`.
- **`dropout_rate`** – `0.5` is the default.

Notes

Use `dropout_rotate_layer.output` and `dropout_rotate_layer.output_shape` from this class.

class `yann.layers.transform.`**`rotate_layer`**(*input*, *input_shape*, *id*, *angle=None*, *borrow=True*, *verbose=2*)

This is a rotate layer. This takes a layer and an angle (rotation normalized in [0,1]) as input and rotates the batch of images by the specified rotation parameter.

Parameters

- **input** – An input `theano.tensor` variable. Even `theano.shared` will work as long as they are in the following shape `mini_batch_size, channels, height, width`
- **verbose** – similar to the rest of the toolbox.
- **input_shape** – `(mini_batch_size, input_size)`
- **angle** – value from [0,1]
- **borrow** – `theano` borrow, typically `True`.
- **input_params** – Supply params or initializations from a pre-trained system.

`batch_norm` - Batch normalization layer classes

The file `yann.layers.batch_norm.py` contains the definition for the batch norm layers. Batch norm can by default be applied to convolution and fully connected layers by sullying an argument `batch_norm = True`, in the layer arguments. But this in-built method applies batch norm prior to layer activation. Some architectures including ResNet involves batch norms after the activations of the layer. Therefore there is a need for an independent batch norm layer that simply applies batch norm for some outputs. The layers in this module can do that.

There are four classes in this file. Two for one-dimensions and two for two-dimnensions.

Todo

- Need to the deconvolutional-unpooling layer.
- Something is still not good about the convolutional batch norm layer.

---

class `yann.layers.batch_norm.`**batch_norm_layer_1d**(*input*, *input_shape*, *id*, *rng=None*, *borrow=True*, *input_params=None*, *verbose=2*)

This class is the typical 1D batchnorm layer. It is called by the `add_layer` method in network class.

Parameters

- **input** – An input `theano.tensor` variable. Even `theano.shared` will work as long as they are in the following shape `mini_batch_size, height, width, channels`
- **verbose** – similar to the rest of the toolbox.
- **input_shape** – `(mini_batch_size, channels, height, width)`
- **rng** – typically `numpy.random`.
- **borrow** – `theano` borrow, typicall `True`.
- **input_params** – Supply params or initializations from a pre-trained system.

class `yann.layers.batch_norm.`**batch_norm_layer_2d**(*input*, *input_shape*, *id*, *rng=None*, *borrow=True*, *input_params=None*, *verbose=2*)

This class is the typical 2D batchnorm layer. It is called by the `add_layer` method in network class.

Parameters

- **input** – An input `theano.tensor` variable. Even `theano.shared` will work as long as they are in the following shape `mini_batch_size, height, width, channels`

- **verbose** – similar to the rest of the toolbox.
- **input_shape** – (mini_batch_size, channels, height, width)
- **rng** – typically numpy.random.
- **borrow** – theano borrow, typicall True.
- **input_params** – Supply params or initializations from a pre-trained system.

class yann.layers.batch_norm.**dropout_batch_norm_layer_1d**(*input*, *input_shape*, *id*, *rng=None*, *borrow=True*, *input_params=None*, *dropout_rate=0*, *verbose=2*)

This class is the typical 1D batchnorm layer. It is called by the add_layer method in network class.

Parameters

- **input** – An input theano.tensor variable. Even theano.shared will work as long as they are in the following shape mini_batch_size, height, width, channels
- **verbose** – similar to the rest of the toolbox.
- **input_shape** – (mini_batch_size, channels, height, width)
- **borrow** – theano borrow, typicall True.
- **dropout_rate** – bernoulli probabilty to dropoutby
- **input_params** – Supply params or initializations from a pre-trained system.

class yann.layers.batch_norm.**dropout_batch_norm_layer_2d**(*input*, *input_shape*, *id*, *rng=None*, *borrow=True*, *input_params=None*, *dropout_rate=0*, *verbose=2*)

This class is the typical 2D batchnorm layer. It is called by the add_layer method in network class.

Parameters

- **input** – An input theano.tensor variable. Even theano.shared will work as long as they are in the following shape mini_batch_size, height, width, channels
- **verbose** – similar to the rest of the toolbox.
- **input_shape** – (mini_batch_size, channels, height, width)
- **borrow** – theano borrow, typicall True.
- **dropout_rate** – bernoulli probabilty to dropoutby
- **input_params** – Supply params or initializations from a pre-trained system.

modules - Modules that are external to the network which the network uses

The module yann.modules contains the definition for the network modules. It contains various modules including:

- visualizer is used to produce network visualizations. It will take the activities, filters and data from a network and produce activations.
- resultor is used to save down network results.

- `optimizer` is the backbone of the SGD and optimization.
- `dataset` is the module that creates, loads, caches and feeds data to the network.

`optimizer` - optimizer class

The file `yann.modules.optimizer.py` contains the definition for the optimizer:

class `yann.modules.optimizer.`**`optimizer`**(*optimizer_init_args*, *verbose=1*)

Optimizer is an important module of the toolbox. Optimizer creates the protocols required for learning. `yann`'s optimizer supports the following optimization techniques:

•Stochastic Gradient Descent

•AdaGrad [1]

•RmsProp [2]

•Adam [3]

•Adadelta [4]

Optimizer also supports the following momentum techniques:

•Polyak [5]

•Nesterov [6]

Parameters

- **`verbose`** – Similar to any 3-level verbose in the toolbox.
- **`optimizer_init_args`** – `optimizer_init_args` is a dictionary like:

```
optimizer_params =  {
    "momentum_type"   : <option>  'false' <no momentum>, 'polyak',
↪'nesterov'.
                        Default value is 'false'
    "momentum_params" : (<option in range [0,1]>, <option in range␣
↪[0,1]>, <int>)
                        (momentum coeffient at start,at end,
                        at what epoch to end momentum increase)
                        Default is the tuple (0.5, 0.95,50)
    "optimizer_type" : <option>, 'sgd', 'adagrad', 'rmsprop', 'adam
↪'.
                       Default is 'sgd'
    "id"         : id of the optimizer
            }
```

Returns Optimizer object

Return type *yann.modules.optimizer*

`calculate_gradients`(*params*, *objective*, *verbose=1*)

This method initializes the gradients.

Parameters

- **`params`** – Supply learnable active parameters of a network.
- **`objective`** – supply a theano graph connecting the params to a loss
- **`verbose`** – Just as always

Notes

Once this is setup, `optimizer.gradients` are available

create_updates(*verbose=1*)
This basically creates all the updates and update functions which trainers can iterate upon.

Parameters **verbose** – Just as always

datastream - datastream class

The file `yann.modules.datastream.py` contains the definition for the datastream:

class `yann.modules.datastream.`**datastream**(*dataset_init_args*, *borrow=True*, *verbose=1*)
This module initializes the dataset to the network class and provides all dataset related functionalities. It also provides for dynamically loading and caching dataset batches. :mod: `add_layer` will use this to initialize.

Parameters

- **dataset_init_args** – Is a dictionary of the form:
- **borrow** – Theano's borrow. Default value is `True`.

```
dataset_init_args = {
            "dataset":  <location>
            "svm"    :  False or True
                 ``svm`` if ``True``, a one-hot label set will␣
↪also be setup.
            "n_classes": <int>
                ``n_classes`` if ``svm`` is ``True``, we need to␣
↪know how
                 many ``n_classes`` are present.
            "id": id of the datastream
    }
```

- **verbose** – Similar to verbose throughout the toolbox.

Returns A dataset module object that has the details of loader and other things.

Return type *dataset*

Todo

•Datastream should work with Fuel perhaps ?

•Support HDf5 perhaps

---

initialize_dataset(*verbose=1*)
Load the initial training batch of data on to `data_x` and `data_y` variables and create shared memories.

Todo

I am assuming that training has the largest number of data. This is immaterial when caching but during set_data routine, I need to be careful.

Parameters **verbose** – Toolbox style verbose.

load_data(*type='train'*, *batch=0*, *verbose=2*)
Will load the data from the file and will return the data. The important thing to note is that all the datasets in :mod: `yann` all require a `y` or a variable to predict. In case of auto-encoder for instance, the thing to predict is the image itself. Setup dataset thusly.

Parameters

- **type** – `train`, `test` or `valid`. default is `train`
- **batch** – Supply an integer
- **verbose** – Simliar to verbose in toolbox.

Todo

Create and load dataset for type = 'x'

Returns `data_x, data_y`

Return type numpy.ndarray

one_hot_labels(*y*, *verbose=1*)
Function takes in labels and returns a one-hot encoding. Used for max-margin loss. :param y: Labels to be encoded.n_classes :param verbose: Typical as in the rest of the toolbox.

Notes

`self.n_classes`: Number of unique classes in the labels.

This could be found out using the following: .. code-block: python

import numpy n_classes = len(numpy.unique(y))

This might be potentially dangerous in case of cached dataset. Although this is the default if `n_classes` is not provided as input to this module, I discourage anyone from using this.

Returns one-hot encoded label list.

Return type numpy ndarray

set_data(*type='train'*, *batch=0*, *verbose=2*)
This can work only after network is cooked.

Parameters

- **batch** – which batch of data to load and set
- **verbose** – as usual

`visualizer` - visualizer class

The file `yann.modules.visualizer.py` contains the definition for the visualizer:

`yann.modules.visualizer.`**save_images**(*imgs*, *prefix*, *is_color*, *verbose=2*)
This functions produces a visualiation of the filters.

Parameters

- **imgs** – images of shape .. [num_imgs, height, width, channels] Note the change in order. it can also be in lots of other shapes and they will be reshaped and saved as images.
- **prefix** – address to save the image to
- **is_color** – If the image is color or not. True only if image shape is of color images.
- **verbose** – As Always

class yann.modules.visualizer.**visualizer**(*visualizer_init_args*, *verbose=2*)

Visualizer saves down images to visualize. The initilizer only initializes the directories for storing visuals. Three types of visualizations are saved down:

•filters of each layer

•activations of each layer

•raw images to check the activations against

Parameters

- **verbose** – Similar to any 3-level verbose in the toolbox.
- **visualizer_init_args** – visualer_params is a dictionary of the form:

```
visualizer_init_args = {
    "root"       : <location to save the visualizations at>,
    "frequency"  : <integer>, after how many epochs do you need to
                    visualize. Default value is 1
    "sample_size": <integer, prefer squares>, simply save down␣
↪random
                    images from the datasets saves down␣
↪activations for the
                    same images also. Default value is 16
    "rgb_filters": <bool> flag. if True a 3D-RGB rendition of the␣
↪CNN
                    filters is rendered. Default value is False.
    "debug_functions" : <bool> visualize train and test and other␣
↪theano functions.
                        default is False. Needs pydot and dv2viz␣
↪to be installed.
    "debug_layers" : <bool> Will print layer activities from input␣
↪to that layer
                     output. ( this is almost always useless␣
↪because test debug
                     function will combine all these layers and␣
↪print directly.)
    "id"         : id of the visualizer
                }
```

Returns A visualizer object.

Return type *yann.modules.visualizer*

initialize(*batch_size*, *verbose=2*)

Function that cooks the visualizer for some dataset.

Parameters

- **batch_size** – form dataset
- **verbose** – as always

theano_function_visualizer(*function*, *short_variable_names=False*, *format='pdf'*, *verbose=2*)
This basically prints a visualization of any theano function using the in-built theano visualizer. It will save both a interactive html file and a plain old png file. This is just a wrapper to theano's visualization tools.

Parameters

- **function** – theano function to print
- **short_variable_names** – If True will print variables in short.
- **format** – Any pydot supported format. Default is 'pdf'
- **verbose** – As usual.

visualize_activities(*layer_activities*, *epoch*, *index=0*, *verbose=2*)
This method saves down all activities.

Parameters

- **layer_activities** – network's layer_activities as created
- **epoch** – what epoch are we running currently.
- **verbose** – as always

visualize_filters(*layers*, *epoch*, *index=0*, *verbose=2*)
This method saves down all activities.

Parameters

- **layers** – network's layer dictionary
- **epoch** – what epoch are we running currently.
- **verbose** – as always

visualize_images(*imgs*, *loc=None*, *verbose=2*)
Visualize the images in the dataset. Assumes that the data in the tensor variable imgs is in shape (batch_size, height, width, channels). Assumes that batchsize does not change.

Parameters

- **imgs** – tensor of data
- **verbose** – as usual.

resultor - resultor class

The file `yann.modules.resultor.py` contains the definition for the resultor:

class `yann.modules.resultor.`**resultor**(*resultor_init_args*, *verbose=1*)
Resultor of the network saves down resultor. The initilizer initializes the directories for storing results.

Parameters

- **verbose** – Similar to any 3-level verbose in the toolbox.
- **resultor_init_args** – `resultor_init_args` is a dictionary of the form

```
resultor_init_args    =    {
    "root"      : "<root directory to save stuff inside>",
    "results"   : "<results_file_name>.txt",
    "errors"    : "<error_file_name>.txt",
    "costs"     : "<cost_file_name>.txt",
```

```
    "learning_rate" : "<learning_rate_file_name>.txt"
    "momentum"  : <momentum_file_name>.txt
    "visualize" : <bool>
    "save_confusion" : <bool>
    "id"        : id of the resultor
                }
```

While the filenames are optional, `root` must be provided. If a particular file is not provided, that value will not be saved.

Returns A resultor object

Return type *yann.modules.resultor*

Todo

Remove the input file names, assume file names as default.

---

print_confusion(*epoch=0*, *train=None*, *valid=None*, *test=None*, *verbose=2*)

This method will print the confusion matrix down in files.

Parameters

- **epoch** – This is used merely to create a directory for each epoch so that there is a copy.
- **train** – training confusion matrix as gained by the validate method.
- **valid** – validation confusion amtrix as gained by the validate method.
- **test** – testing confusion matrix as gained by the test method.
- **verbose** – As usual.

process_results(*cost*, *lr*, *mom*, *params=None*, *verbose=2*)

This method will print results and also write them down in the appropriate files.

Parameters

- **cost** – Cost, is a float
- **lr** – Learning Rate, is a float
- **mom** – Momentum, is a float.

update_plot(*verbose=2*)

Todo

This method should update the open plots with costs and other values. Ideally, a browser based system should be implemented, such as using mpl3d or using bokeh. This system should open opne browser where it should update realtime the cost of training, validation and testing accuracies per epoch, display the visualizations of filters, some indication of the weight of gradient trained, confusion matrices, learning rate and momentum plots etc.

`core` - Core module contains the fundamentals operations of the toolbox

The module `core` contains the work horse of the `yann` toolbox. These are very fundamental operations that are not needed to be known by or used by a user, but other modules in the toolbox will use these constantly.

conv - Definitions for all convolution functions.

The file yann.core.conv.py contains the definition for all the convolution functions available. yann.core.conv.py is one file that contains all the convolution operators. It contains two functions for performing either 2d convolution (conv2d) or 3d convolution (conv3d).

These functions shall be called by every convolution layer from yann.layers.py

---

Todo

- Add 3D convolution support from theano.
- Add Masked convolution support.

---

class yann.core.conv.**convolver_2d**(*input*, *filters*, *subsample*, *filter_shape*, *image_shape*, *border_mode='valid'*, *verbose=1*)

Class that performs convolution

This class basically performs convolution. These ouputs can be probed using the convolution layer if needed. This keeps things simple.

Parameters

- **input** – This variable should either thenao.tensor4 (theano.matrix reshaped also works) variable or an output from a pervious layer which is a theano.tensor4 convolved with a theano.shared. The input should be of shape (batchsize, channels, height, width). For those who have tried pylearn2 or such, this is called bc01 format.
- **fitlers** – This variable should be theano.shared variables of filter weights could even be a filter bank. filters should be of shape (nchannels, nkerns, filter_height, filter_width). nchannles is the number of input channels and nkerns is the number of kernels or output channels.
- **subsample** – Stride Tuple of (int, int).
- **filter_shape** – This variable should be a tuple or an array: [nkerns, nchannles, filter_height, filter_width]
- **image_shape** – This variable should a tuple or an array: [batchsize, channels, height, width] image_shape[1] must be equal to filter_shape[1]
- **border_mode** – The input to this can be either 'same' or other theano defaults

Notes

•conv2d.out output, Output that could be provided as output to the next layer or to other convolutional layer options. The size of the outut depends on border mode and subsample operation performed.

•conv2d.out_shp: (int, int), A tuple (height, width) of all feature maps

The options for border_mode input which at the moment of writing this doc are

•'valid' - apply filter wherever it completely overlaps with the input. Generates output of shape input shape - filter shape + 1

•'full'- apply filter wherever it partly overlaps with the input. Generates output of shape input shape + filter shape - 1

- `'half'`: pad input with a symmetric border of `filter rows // 2` rows and `filter columns // 2` columns, then perform a valid convolution. For filters with an odd number of rows and columns, this leads to the output shape being equal to the input shape.
- `<int>`: pad input with a symmetric border of zeros of the given width, then perform a valid convolution.
- `(<int1>, <int2>)`: pad input with a symmetric border of `int1` rows and `int2` columns, then perform a valid convolution.

Refer to theano documentation's convolution page for more details on this. Basically cuDNN is used for `same` because at the moment of writing this funciton, `theano.conv2d` doesn't support''same'' convolutions on the GPU. For everything else, `theano` default will be used.

Todo

Implement `border_mode = 'same'` for libgpuarray backend. As of now only supports CUDA backend.

Need to something about this. With V0.10 of theano, I cannot use cuda.dnn for same convolution.

---

class `yann.core.conv.`**`deconvolver_2d`**(*input*, *filters*, *subsample*, *filter_shape*, *image_shape*, *output_shape*, *border_mode='valid'*, *verbose=1*)

class that performs deconvolution

This class basically performs convolution.

Parameters

- **`input`** – This variable should either `thenao.tensor4` (`theano.matrix` reshaped also works) variable or an output from a pervious layer which is a `theano.tensor4` convolved with a `theano.shared`. The input should be of shape `(batchsize, channels, height, width)`. For those who have tried `pylearn2` or such, this is called bc01 format.
- **`fitlers`** – This variable should be `theano.shared` variables of filter weights could even be a filter bank. `filters` should be of shape `(nchannels, nkerns, filter_height, filter_width)`. `nchannles` is the number of input channels and `nkerns` is the number of kernels or output channels.
- **`subsample`** – Stride Tuple of `(int, int)`.
- **`filter_shape`** – This variable should be a tuple or an array: `[nkerns, nchannles, filter_height, filter_width]`
- **`image_shape`** – This variable should a tuple or an array: `[batchsize, channels, height, width]` `image_shape[1]` must be equal to `filter_shape[1]`
- **`output_shape`** – Request a size of output of image required. This variable should a tuple.
- **`border_mode`** – The input to this can be either `'same'` or other theano defaults

Notes

- `conv2d.out` output, Output that could be provided as output to the next layer or to other convolutional layer options. The size of the outut depends on border mode and subsample operation performed.
- `conv2d.out_shp`: (`int`, `int`), A tuple (height, width) of all feature maps

The options for `border_mode` input which at the moment of writing this doc are

- `'valid'` - apply filter wherever it completely overlaps with the input. Generates output of shape `input shape - filter shape + 1`
- `'full'`- apply filter wherever it partly overlaps with the input. Generates output of shape `input shape + filter shape - 1`
- `'half'`: pad input with a symmetric border of `filter rows // 2` rows and `filter columns // 2` columns, then perform a valid convolution. For filters with an odd number of rows and columns, this leads to the output shape being equal to the input shape.
- `<int>`: pad input with a symmetric border of zeros of the given width, then perform a valid convolution.
- `(<int1>, <int2>)`: pad input with a symmetric border of `int1` rows and `int2` columns, then perform a valid convolution.

Refer to theano documentation's convolution page for more details on this. Basically cuDNN is used for `same` because at the moment of writing this funciton, `theano.conv2d` doesn't support''same'' convolutions on the GPU. For everything else, `theano` default will be used.

Todo

Implement `border_mode = 'same'` and full for libgpuarray backend. As of now only supports CUDA backend.

Need to something about this. With V0.10 of theano, I cannot use cuda.dnn for same convolution.

Right now deconvolution works only with `border_mode = 'valid'`

`pool` - Definitions for all Pooling functions.

The file `yann.core.pool.py` contains the definition for all the Pooling functions available.

Todo

- Need to support max_rand_pool and rand_pool

---

class `yann.core.pool.`**`pooler_2d`**(*input*, *ds*, *img_shp*, *mode='max'*, *ignore_border=True*, *verbose=1*)

function that performs pooling

Parameters

- **`input`** – This variable should either `thenao.tensor4` (`theano.matrix` reshaped also works) variable or an output from a pervious layer which is a `theano.tensor4` convolved with a `theano.shared`. The input should be of shape `(batchsize, channels, height, width)`. For those who have tried pylearn2 or such, this is called bc01 format.
- **`img_shp`** – This variable should a tuple or an array: `[batchsize, channels, height, width]`
- **`ds`** – tuple of pool sizes for rows and columns. `(pool height, pool width)`
- **`mode`** – {'max', 'sum', 'mean', 'max_same_size'} Operation executed on each window. *max* and *sum*
  - if `max_same_size` we do maxpooling with output is the same size.
  - if `max` we do do maxpooling with output being downsampled. Output size will be `(batchsize, channels, height/ds[0], width/ds[1])`.

  - if `mean` we do do meanpooling with output being downsampled. Output size will be `(batchsize, channels, height/ds[0], width/ds[1])`.
  - if `sum` we do do sum pooling with output being downsampled. Output size will be `(batchsize, channels, height/ds[0], width/ds[1])`.
- **ignore_border** – (default is `False`) Consider theano's documentation. It is directly supplied to theano's pool module.

`activations` - Definitions for all activations functions.

The file `yann.core.activations.py` contains the definition for all the activation functions available.

You can import all these functions and supply the fuctions as arguments to functions that use `activation` variable as an input. Refer to the mnist example in the modelzoo for how to do this. It contains various activations as defined below:

`yann.core.activations.`**`Abs`**(*x*)

Absolute value Units.

Applies point-wise absolute value to the input supplied.

Parameters **x** – could be a `theano.tensor` or a `theano.shared` or `numpy` arrays or `python lists`.

Returns returns a absolute output of the same shape as the input.

Return type same as input

`yann.core.activations.`**`Elu`**(*x*, *alpha=1*)

Exponential Linear Units.

Applies point-wise ela to the input supplied. `alpha` is defualt to `0`. Supplying a value to `alpha` would make this a leay Elu.

Notes

Reference :Clevert, Djork-Arne, Thomas Unterthiner, and Sepp Hochreiter. "Fast and accurate deep network learning by exponential linear units (elus)." arXiv preprint arXiv:1511.07289 (2015).

Parameters

- **x** – could be a `theano.tensor` or a `theano.shared` or `numpy` arrays or `python lists`.
- **alpha** – should be a `float`. Default is `1`.

Returns returns a point-wise rectified output.

Return type same as input

`yann.core.activations.`**`Maxout`**(*x*, *maxout_size*, *input_size*, *type='maxout'*, *dimension=1*)

Function performs the maxout activation. You can import all these functions and supply the fuctions as arguments to functions that use `activation` variable as an input. Refer to the mnist example in the modelzoo for how to do this.

Parameters

- **x** – could be a `theano.tensor` or a `theano.shared` or `numpy` arrays or `python lists`. Size of the argument must strictly be windowed runnable through `stride`. Second dimension must be the channels to maxout from
- **maxout_size** – is the size of the window to stride through
- **input_size** – is number of nodes in the input
- **dimension** – If `1` perform MLP layer maxout, input must be two dimensional. If `2` perform CNN layer maxout, input must be four dimensional.
- **type** – If `maxout` perform, [1] If `meanout` or `mixedout` perform, meanout or mixed out respectively from [2]

Returns

1. **theano.tensor4 output, Output that could be provided**

   as output to the next layer or to other convolutional layer options. the size of the output depends on border mode and subsample operation performed.

   (a) `tuple`, Number of feature maps after maxout is applied

Return type `theano.tensor4`

`yann.core.activations.`**ReLU**(*x*, *alpha=0*)

Rectified Linear Units.

Applies point-wise rectification to the input supplied. `alpha` is defualt to `0`. Supplying a value to `alpha` would make this a leay ReLU.

Notes

Reference: Nair, Vinod, and Geoffrey E. Hinton. "Rectified linear units improve restricted boltzmann machines." Proceedings of the 27th International Conference on Machine Learning (ICML-10). 2010.

Parameters

- **x** – could be a `theano.tensor` or a `theano.shared` or `numpy` arrays or `python lists`.
- **alpha** – should be a `float`.

Returns returns a point-wise rectified output.

Return type same as input

`yann.core.activations.`**Sigmoid**(*x*)

Sigmoid Units.

Applies point-wise sigmoid to the input supplied.

Parameters **x** – could be a `theano.tensor` or a `theano.shared` or `numpy` arrays or `python lists`.

Returns returns a point-wise sigmoid output of the same shape as the input.

Return type same as input

yann.core.activations.**Softmax**(*x*, *temp=1*)
Softmax Units.

Applies row-wise softmax to the input supplied.

Parameters

- **x** – could be a theano.tensor or a theano.shared or numpy arrays or python lists.
- **temp** – temperature of type float. Mainly used during distillation, normal softmax prefer T=1.

Notes

Refer [3] for details.

Returns returns a row-wise softmax output of the same shape as the input.

Return type same as input

yann.core.activations.**Squared**(*x*)
Squared Units.

Applies point-wise squaring to the input supplied.

Parameters **x** – could be a theano.tensor or a theano.shared or numpy arrays or python lists.

Returns returns a squared output of the same shape as the input.

Return type same as input

yann.core.activations.**Tanh**(*x*)
Tanh Units.

Applies point-wise hyperbolic tangent to the input supplied.

Parameters **x** – could be a theano.tensor or a theano.shared or numpy arrays or python lists.

Returns returns a point-wise hyperbolic tangent output.

Return type same as input

errors - Definitions for all point-wise error functions.

The file yann.core.errors.py contains the definition for all the point-wise error functions available.

yann.core.errors.**cross_entropy**(*a*, *b*)
This function produces a point-wise cross entropy error between a and b

Parameters

- **a** – first input
- **b** – second input

Returns Computational graph with the error.

Return type theano shared variable

yann.core.errors.**l1**(*a*, *b*)
This function produces a point-wise L1 error between a and b

Parameters

- **a** – first input
- **b** – second input

Returns Computational graph with the error.

Return type theano shared variable

yann.core.errors.**rmse**(*a*, *b*)
This function produces a point-wise root mean squared error error between a and b

Parameters

- **a** – first input
- **b** – second input

Returns Computational graph with the error.

Return type theano shared variable

operators - Definitions for all basic operators written for yann .

The file yann.core.operators.py contains the definition for all the operators written specifically for yann. Typically yann's fundamental operators come from theano or numpy. This is needed only for explicit operators not available clearly elsewhere.

yann.core.operators.**copy_params**(*source*, *destination*, *borrow=True*, *verbose=2*)
Internal function that copies paramters maintaining theano shared nature.

Parameters

- **source** – Source
- **destination** – destination

Notes

Was using deep copy to do this. This seems faster. But can I use theano.clone ?

utils - utilities that can be used as enhancement for the toolbox's functionality

Many additional utilities are provided with this toolbox. They are divided into different modules and written in many files. Below is an index of all files and their documentations.

dataset - provides a nice port to benchmark and matlab-based datasets

The file yann.utils.dataset.py contains the definition for the dataset ports. It contains support to various benchmark datasets through skdata. There is also support to a dataset that can be imported from matlab.

Todo

- None of the PASCAL dataset retrievers from skdata is working. This need to be coded in.
- Need a method to create dataset from a directory of images. - prepare for imagenet and coco.
- See if support can be made for fuel.

yann.utils.dataset.**create_shared_memory_dataset**(*data_xy*, *borrow=True*, *verbose=1*, ***kwargs*)

This function creates a shared theano memory to be used for dataset purposes.

Parameters

- **data_xy** – `[data_x, data_y]` that will be assigned to `shared_x` and `shared_y` on output.
- **borrow** – default value is `True`. This is a theano shared memory type variabe.
- **verbose** – Similar to verbose everywhere else.
- **svm** – default is `False`. If `True`, we also return a `shared_svm_y` for max-margin type last layer.

Returns

shared_x, shared_y is svm is `False`. If not, ‘‘shared_x, shared_y, shared_svm_y‘‘

Return type theano.shared

yann.utils.dataset.**download_data**(*url*, *location*)

yann.utils.dataset.**load_cifar100**()

Function that downloads the cifar 100 dataset and returns the dataset in full

TODO: Need to implement this.

yann.utils.dataset.**load_data_mat**(*height*, *width*, *channels*, *location*, *batch=0*, *type_set='train'*, *load_z=False*)

Use this code if the data was created in matlab in the right format and needed to be loaded. The way to create is to have variables `x, y, z` with `z` being an optional data to load. `x` is assumed to be the data in matrix `double` format with rows being each image in vectorized fashion and `y` is assumed to be lables in `int` or `double`.

The files are stored in the following format: `loc/type/batch_0.mat`. This code needs scipy to run.

Parameters

- **height** – The height of each image in the dataset.
- **width** – The width of each image in the dataset.
- **channels** – `3` if RGB, `1` if grayscale and so on.
- **location** – Location of the dataset.
- **batch** – if multi batch, then how many batches of data is present if not use `1`

Returns Tuple *(data_x, data_y)* if requested, also *(data_x,data_y,data_z)*

Return type float32 tuple

Todo

Need to add preprocessing in this.

---

yann.utils.dataset.**load_images_only**(*batch_size*, *location*, *n_train_images*, *n_test_images*, *n_valid_images*, *rand_perm*, *batch=1*, *type_set='train'*, *height=218*, *width=178*, *channels=3*, *verbose=False*)

Function that downloads the dataset and returns the dataset in full.

Parameters

- **mini_batch_size** – What is the size of the batch.
- **n_train_images** – number of training images.
- **n_test_images** – number of testing images.
- **n_valid_images** – number of validating images.
- **rand_perm** – Create a random permutation list of images to be sampled to batches.
- **type_set** – What dataset you need, test, train or valid.
- **height** – Height of the image
- **width** – Width of the image.
- **verbose** – similar to dataset.

Returns `data_x`

Return type list

yann.utils.dataset.**load_skdata_caltech101**(*batch_size*, *n_train_images*, *n_test_images*, *n_valid_images*, *rand_perm*, *batch=1*, *type_set='train'*, *height=256*, *width=256*, *verbose=False*)

Function that downloads the dataset from skdata and returns the dataset in part

Parameters

- **batch_size** – What is the size of the batch.
- **n_train_images** – number of training images.
- **n_test_images** – number of testing images.
- **n_valid_images** – number of validating images.
- **rand_perm** – Create a random permutation list of images to be sampled to batches.
- **type_set** – What dataset you need, test, train or valid.
- **height** – Height of the image
- **width** – Width of the image.
- **verbose** – similar to dataset.

Todo

This is not a finished function.

Returns `[(train_x, train_y, train_y),(valid_x, valid_y, valid_y), (test_x, test_y, test_y)]`

Return type list

yann.utils.dataset.**load_skdata_caltech256**(*batch_size*, *n_train_images*, *n_test_images*, *n_valid_images*, *rand_perm*, *batch=1*, *type_set='train'*, *height=256*, *width=256*, *verbose=False*)

Function that downloads the dataset from skdata and returns the dataset in part

Parameters

- **mini_batch_size** – What is the size of the batch.
- **n_train_images** – number of training images.
- **n_test_images** – number of testing images.
- **n_valid_images** – number of validating images.
- **rand_perm** – Create a random permutation list of images to be sampled to batches.
- **type_set** – What dataset you need, test, train or valid.
- **height** – Height of the image
- **width** – Width of the image.
- **verbose** – similar to dataset.

Todo

This is not a finished function.

Returns `[(train_x, train_y, train_y),(valid_x, valid_y, valid_y), (test_x, test_y, test_y)]`

Return type list

yann.utils.dataset.**load_skdata_cifar10**()
Function that downloads the dataset from skdata and returns the dataset in full

Returns `[(train_x, train_y, train_y),(valid_x, valid_y, valid_y), (test_x, test_y, test_y)]`

Return type list

yann.utils.dataset.**load_skdata_mnist**()
Function that downloads the dataset from skdata and returns the dataset in full

Returns `[(train_x, train_y, train_y),(valid_x, valid_y, valid_y), (test_x, test_y, test_y)]`

Return type list

yann.utils.dataset.**load_skdata_mnist_bg_images**()
Function that downloads the dataset from skdata and returns the dataset in full

Returns `[(train_x, train_y, train_y),(valid_x, valid_y, valid_y), (test_x, test_y, test_y)]`

Return type list

yann.utils.dataset.**load_skdata_mnist_bg_rand**()
Function that downloads the dataset from skdata and returns the dataset in full

Returns `[(train_x, train_y, train_y),(valid_x, valid_y, valid_y), (test_x, test_y, test_y)]`

Return type list

yann.utils.dataset.**load_skdata_mnist_noise1**()
Function that downloads the dataset from skdata and returns the dataset in full

Returns `[(train_x, train_y, train_y),(valid_x, valid_y, valid_y), (test_x, test_y, test_y)]`

Return type list

yann.utils.dataset.**load_skdata_mnist_noise2**()
Function that downloads the dataset from skdata and returns the dataset in full

Returns [(train_x, train_y, train_y),(valid_x, valid_y, valid_y), (test_x, test_y, test_y)]

Return type list

yann.utils.dataset.**load_skdata_mnist_noise3**()
Function that downloads the dataset from skdata and returns the dataset in full

Returns [(train_x, train_y, train_y),(valid_x, valid_y, valid_y), (test_x, test_y, test_y)]

Return type list

yann.utils.dataset.**load_skdata_mnist_noise4**()
Function that downloads the dataset from skdata and returns the dataset in full

Returns [(train_x, train_y, train_y),(valid_x, valid_y, valid_y), (test_x, test_y, test_y)]

Return type list

yann.utils.dataset.**load_skdata_mnist_noise5**()
Function that downloads the dataset from skdata and returns the dataset in full

Returns [(train_x, train_y, train_y),(valid_x, valid_y, valid_y), (test_x, test_y, test_y)]

Return type list

yann.utils.dataset.**load_skdata_mnist_noise6**()
Function that downloads the dataset from skdata and returns the dataset in full

Returns [(train_x, train_y, train_y),(valid_x, valid_y, valid_y), (test_x, test_y, test_y)]

Return type list

yann.utils.dataset.**load_skdata_mnist_rotated**()
Function that downloads the dataset from skdata and returns the dataset in full

Returns [(train_x, train_y, train_y),(valid_x, valid_y, valid_y), (test_x, test_y, test_y)]

Return type list

yann.utils.dataset.**load_skdata_mnist_rotated_bg**()
Function that downloads the dataset from skdata and returns the dataset in full

Returns [(train_x, train_y, train_y),(valid_x, valid_y, valid_y), (test_x, test_y, test_y)]

Return type list

yann.utils.dataset.**pickle_dataset**(*loc*, *batch*, *data*)
Function that stores down an object as a pickle file given its filename and obj

Parameters

- **loc** – Provide location to save as a string
- **batch** – provide a batch number to save the file as

- **data** – Pass the data that needs to be picked down. Could also be a tuple

class yann.utils.dataset.**setup_dataset**(*dataset_init_args*, *save_directory='_datasets'*, *verbose=1*, ***kwargs*)

The setup_dataset class is used to create and assemble datasets that are friendly to the Yann toolbox.

Todo

images option for the source. skdata pascal isn't working imagenet dataset and coco needs to be setup.

Parameters

- **dataset_init_args** – is a dictonary of the form:

```
data_init_args = {

    "source" : <where to get the dataset from>
                'pkl' : A theano tutorial style 'pkl' file.
                'skdata' : Download and setup from skdata
                'matlab' : Data is created and is being used from␣
↪Matlab
                'images-only' : Data is created from a directory␣
↪of images. This
                          will be an unsupervised dataset with no␣
↪labels.
    "name" : necessary only for skdata
              supports
                * ``'mnist'``
                * ``'mnist_noise1'``
                * ``'mnist_noise2'``
                * ``'mnist_noise3'``
                * ``'mnist_noise4'``
                * ``'mnist_noise5'``
                * ``'mnist_noise6'``
                * ``'mnist_bg_images'``
                * ``'mnist_bg_rand'``
                * ``'mnist_rotated'``
                * ``'mnist_rotated_bg'``.
                * ``'cifar10'``
                * ``'caltech101'``
                * ``'caltech256'``

          Refer to original paper by Hugo Larochelle [1] for these␣
↪dataset details.

    "location"                  : necessary for 'pkl' and 'matlab'␣
↪and
                                      'images-only'
    "mini_batch_size"           : 500, # some batch size
    "mini_batches_per_batch"    : (100, 20, 20), # trianing,␣
↪testing, validation
    "batches2train"             : 1, # number of files will be␣
↪created.
    "batches2test"              : 1,
    "batches2validate"          : 1,
    "height"                    : 28, # After pre-processing
    "width"                     : 28,
```

```
    "channels"                  : 1 , # color (3) or grayscale (1)␣
↪...

        }
```

- **preprocess_init_args** – provide preprocessing arguments. This is a dictionary:

```
args =  {
    "normalize" : <bool> True for normalize across batches
    "GCN"       : True for global contrast normalization
    "ZCA"       : True, kind of like a PCA representation (not␣
↪fully tested)
    "grayscale" : Convert the image to grayscale
        }
```

- **save_directory** – <string> a location where the dataset is going to be saved.

Notes

Yann toolbox takes datasets in a `.pkl` format. The dataset requires a directory structure such as the following:

```
location/_dataset_XXXXX
|_ data_params.pkl
|_ train
    |_ batch_0.pkl
    |_ batch_1.pkl
    .
    .
    .
|_ valid
    |_ batch_0.pkl
    |_ batch_1.pkl
    .
    .
    .
|_ test
    |_ batch_0.pkl
    |_ batch_1.pkl
    .
    .
    .
```

The location id (`XXXXX`) is generated by this class file. The five digits that are produced is the unique id of the dataset.

The file `data_params.pkl` contains one variable `dataset_args` used by datastream.

dataset_location()
Use this function that return the location of dataset.

yann.utils.dataset.**shuffle**(*data*, *verbose=1*)
Method shuffles the dataset with x and y

Parameters **data** – Either a tuple of a nd array. If tuple, will assume x and y.

Returns Shuffled version of the same.

Return type data

Notes

Obnly tuple works at the moment.

`graph` - provides a nice port to networkx methods related to Yann

The file `yann.utils.graph.py` contains the definition for the networkx ports. If `networkx` was installed, each network class also creates a `networkx.DiGraph` within itself which is accessible through *net = network()*, *net.graph*. In each layer some representative nodes (max limited) will be added to this graph and can be seen at *net.graph.nodes()*. Its attributes will be layer prorperties such as *type*, *output_shape* and so on.

yann.utils.graph has some ports that uses this networkx graph.

This includes: *draw_network* which draws the network.

The documentation follows:

`yann.utils.graph.`**`draw_network`**(*graph*, *filename='network.pdf'*, *show=False*, *verbose=2*)

This is a simple wrapper to the networkx_draw.

Parameters

- **`graph`** – Supply a networkx graph object. NNs are all DiGraphs.
- **`filename`** – what file to save down as. Will add '.png' to the end.
- **`verbose`** – Do I even have to talk about this ?

Notes

Takes any format that networkx plotter takes. This is not ready to be used. Still buggy sometimes. Rudra is working on developing this further internally. This is slow at the moment.

`pickle` - provides a way to save the network' parameters as a pickle file.

The file `yann.utils.pickle.py` contains the definition for the pickle methods. Use pickle method in the file to save the params down as a pickle file. Note that this only saves the parameters down and not the architecture or optimizers or other modules. The id of the layers will also be saved along as dictionary keys so you can use them to create a network.

The documentation follows:

`yann.utils.pickle.`**`load`**(*infile*, *verbose=2*)

This method loads a pickled network and returns the parameters.

Parameters **`infile`** – Filename of the network pickled by this pickle method.

Returns A dictionary of parameters.

Return type params

`yann.utils.pickle.`**`pickle`**(*net*, *filename*, *verbose=2*)

This method saves the weights of all the layers.

Parameters

- **`net`** – A yann network object
- **`filename`** – What is the name of the file to pickle the network as.

- **verbose** – Blah..

yann.utils.pickle.**shared_params**(*params*, *verbose=2*)

This will convert a loaded set of parameters to shared variables that could be passed as `input_params` to the `add_layer` method.

Parameters **params** – List from `get_params` method.

raster - provides a visualization for rasterizing images.

The file `yann.utils.raster.py` contains the definition for the rasters. This code is not mine and is lifted from theano utils. This code is used to remove the dependency on now out-dated pylearn2 library. Yann uses pylearn2's make_viewer to create images that are raster. Migrating to this tile_raster_images from theano tutorials. Obliging License, credit and conditions for Theano Deep Learning Tutorials: This entire file was completely and directly reproduced from the theano deep learning tutorials.

Copyright (c) 2010–2015, Deep Learning Tutorials Development Team All rights reserved. This code is used to remove the dependency on now out-dated pylearn2 library. Yann uses pylearn2's make_viewer to create images that are raster. Migrating to this tile_raster_images from theano tutorials.

Obliging License, credit and conditions for Theano Deep Learning Tutorials: This entire file was completely and directly reproduced from the theano deep learnign tutorials.

LICENSE

yann.utils.raster.**scale_to_unit_interval**(*ndar*, *eps=1e-08*)

Scales all values in the ndarray ndar to be between 0 and 1

yann.utils.raster.**tile_raster_images**(*X*, *img_shape*, *tile_shape*, *tile_spacing=(0, 0)*, *scale_rows_to_unit_interval=True*, *output_pixel_vals=True*)

Transform an array with one flattened image per row, into an array in which images are reshaped and layed out like tiles on a floor. This function is useful for visualizing datasets whose rows are images, and also columns of matrices for transforming those rows (such as the first layer of a neural net). :type X: a 2-D ndarray or a tuple

of 4 channels, elements of which can be 2-D ndarrays or None; :param X: a 2-D array in which every row is a flattened image. :type img_shape: tuple; (height, width) :param img_shape: the original shape of each image :type tile_shape: tuple; (rows, cols) :param tile_shape: the number of images to tile (rows, cols) :param output_pixel_vals: if output should be pixel values (i.e. int8 values) or floats :param scale_rows_to_unit_interval: if the values need to be scaled before being plotted to [0,1] or not :returns: array suitable for viewing as an image. (See:*Image.fromarray*.) :rtype: a 2-d array with same dtype as X.

`special` - contains tools for special types of networks

This module contains tools for special types of networks. Most special types of networks involve inheriting the main network class and re-writing a new class for different types of networks

`gan` - provides a inherited network class for a gan network.

The file `yann.special.gan.py` contains the definition for gan-style network. Any GAN network can be built using this class. It is basically an inherited network from the yann.network file. Support for the implementation from

Goodfellow, Ian, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. "Generative adversarial nets." In Advances in Neural Information Processing Systems, pp. 2672-2680. 2014.

Todo

There seems to be something wrong with the fine-tuning update. Code crashes after a call to _new_era. This needs debugging and fixing.

---

class `yann.special.gan.` **gan** (*verbose=2*, ***kwargs*)

This class is inherited from the network class and has its own methods modified in support of gan networks.

Todo

Sumith Chintala says that its better to seperate generator and dataset when training discriminator. Do that.

in __init__ kwargs = kwargs is not a good option Check its working.

Parameters **as the network class** (*Same*) –

cook (*objective_layers*, *discriminator_layers*, *generator_layers*, *game_layers*, *softmax_layer=None*, *classifier_layers=None*, *optimizer_params=None*, *verbose=2*, ***kwargs*)

This function builds the backprop network, and makes the trainer, tester and validator theano functions. The trainer builds the trainers for a particular objective layer and optimizer.

Parameters

- **optimizer_params** – Supply optimizer_params.
- **datastream** – Supply which datastream to use. Default is the last datastream created.
- **visualizer** – Supply a visualizer to cook with.
- **objective_layers** – Supply a tuple of layer ids of layers that have the objective functions (classification, discriminator, generator)
- **classifier** – supply the classifier layer of the discriminator.
- **discriminator** – supply the discriminator layer of the data stream.

- **generator** – supply the last generator layer.
- **generator_layers** – list or tuple of all generator layers
- **discriminator_layers** – list or tuple of all discriminator layers
- **classifier_layers** – list or tuple of all classifier layers
- **game_layers** – list or tuple of two layers. The first is D(G(z)) and the second is D(x)
- **verbose** – Similar to the rest of the toolbox.

cook_discriminator(*optimizer_params*, *verbose=2*)
This method cooks the real optimizer.

Parameters **verbose** – as always

cook_generator(*optimizer_params*, *verbose=2*)
This method cooks the fake optimizer.

Parameters **verbose** – as always

cook_softmax_optimizer(*optimizer_params*, *verbose=2*)
This method cooks the softmax optimizer.

Parameters **verbose** – as always

initialize_train(*verbose=2*)
Internal function that creates a train methods for the GAN network

Parameters **verbose** – as always

print_status(*epoch*, *verbose=2*)
This function prints the costs of the current epoch, learning rate and momentum of the network at the moment.

Todo

This needs to to go to visualizer.

Parameters

- **verbose** – Just as always.
- **epoch** – Which epoch are we at ?

train(*verbose*, ***kwargs*)
Training function of the network. Calling this will begin training.

Parameters

- **epochs** – `(num_epochs for each learning rate... )` to train Default is `(20, 20)`
- **validate_after_epochs** – 1, after how many epochs do you want to validate ?
- **show_progress** – default is `True`, will display a clean progressbar. If `verbose` is `3` or more - False
- **early_terminate** – `True` will allow early termination.
- **k** – how many discriminator updates for every generator update.

- **learning_rates** – (annealing_rate, learning_rates ...) length must be one more than `epochs` Default is `(0.05, 0.01, 0.001)`
- **save_after_epochs** – 1, Save network after that many epochs of training.
- **pre_train_discriminator** – If you want to pre-train the discriminator to make it stay ahead of the generator for making predictions. This will only train the softmax layer loss and not the fake or real loss.

validate (*epoch=0*, *training_accuracy=False*, *show_progress=False*, *verbose=2*)
Method is use to run validation. It will also load the validation dataset.

Parameters

- **verbose** – Just as always
- **show_progress** – Display progressbar ?
- **training_accuracy** – Do you want to print accuracy on the training set as well ?

`datasets` - provides quick methods to produce common datasets.

The file `yann.special.datasets.py` contains the definition for some methods that can produce quickly some datasets. Some of them include :

- `cook_mnist`
- `cook_cifar10`
- ...

class `yann.special.datasets.`**combine_split_datasets** (*loc*, *verbose=1*, ***kwargs*)
This will combine two split datasets into one.

Todo

Extend it for non-split datasets also.

Parameters

- **loc** – A tuple of a list of locations of two dataset to be blended.
- **verbose** – As always

Notes

At this moment, mini_batches_per_batch and mini_batch_size of both datasets must be the same. This only splits the train data with shot. The test and valid hold both. This is designed for the incremental learning. New labels are created in one shot labels for the second datasets. This does not assume that labels are shared between the two datasets.

combine (*verbose=1*)
Thie method runs the combine.

Parameters **verbose** – As Always

dataset_location ()
Use this function that return the location of dataset.

load_data(*n_batches_1*, *n_batches_2*, *type='train'*, *batch=0*, *verbose=2*)
Will load the data from the file and will return the data. Will supply two batches one from each set respectively.

Parameters

- **type** – `train`, `test` or `valid`. default is `train`
- **batch** – Supply an integer
- **n_batches_1** – Number of batches in dataset 1
- **n_batches_2** – Number of batches in dataset 2
- **verbose** – Simliar to verbose in toolbox.

Todo

Create and load dataset for type = 'x'

Returns `data_x, data_y`

Return type numpy.ndarray

save_data(*data_x*, *data_y*, *type='train'*, *batch=0*, *verbose=2*)
Saves down a batch of data.

yann.special.datasets.**cook_caltech101**(*verbose=1*, ***kwargs*)
Wrapper to cook cifar10 dataset. Will take as input,

Parameters

- **save_directory** – which directory to save the cooked dataset onto.
- **dataset_parms** – default is the dictionary. Refer to `setup_dataset`
- **preprocess_params** – default is the dictionary. Refer to `setup_dataset`

yann.special.datasets.**cook_caltech256**(*verbose=1*, ***kwargs*)
Wrapper to cook cifar10 dataset. Will take as input,

Parameters

- **save_directory** – which directory to save the cooked dataset onto.
- **dataset_parms** – default is the dictionary. Refer to `setup_dataset`
- **preprocess_params** – default is the dictionary. Refer to `setup_dataset`

yann.special.datasets.**cook_celeba_normalized_zero_mean**(*verbose=1*, *location='_data/celebA'*, ***kwargs*)
Wrapper to cook Celeb-A dataset in preparation for GANs. Will take as input,

Parameters

- **location** – Location where celebA was downloaded using `yann.specials.datasets.download_celebA`
- **save_directory** – which directory to save the cooked dataset onto.
- **dataset_parms** – default is the dictionary. Refer to `setup_dataset`
- **preprocess_params** – default is the dictionary. Refer to `setup_dataset`

yann.special.datasets.**cook_cifar10**(*verbose=1*, ***kwargs*)
Wrapper to cook cifar10 dataset. Will take as input,

Parameters

- **save_directory** – which directory to save the cooked dataset onto.
- **dataset_parms** – default is the dictionary. Refer to `setup_dataset`
- **preprocess_params** – default is the dictionary. Refer to `setup_dataset`

yann.special.datasets.**cook_cifar10_normalized**(*verbose=1*, ***kwargs*)
Wrapper to cook cifar10 dataset. Will take as input,

Parameters

- **save_directory** – which directory to save the cooked dataset onto.
- **dataset_parms** – default is the dictionary. Refer to `setup_dataset`
- **preprocess_params** – default is the dictionary. Refer to `setup_dataset`

yann.special.datasets.**cook_cifar10_normalized_zero_mean**(*verbose=1*, ***kwargs*)
Wrapper to cook cifar10 dataset. Will take as input,

Parameters

- **save_directory** – which directory to save the cooked dataset onto.
- **dataset_parms** – default is the dictionary. Refer to `setup_dataset`
- **preprocess_params** – default is the dictionary. Refer to `setup_dataset`

yann.special.datasets.**cook_mnist**(*verbose=1*, ***kwargs*)
Wrapper to cook mnist dataset. Will take as input,

Parameters

- **save_directory** – which directory to save the cooked dataset onto.
- **dataset_parms** – default is the dictionary. Refer to `setup_dataset`
- **preprocess_params** – default is the dictionary. Refer to `setup_dataset`

Notes

By default, this will create a dataset that is not mean-subtracted.

yann.special.datasets.**cook_mnist_multi_load**(*verbose=1*, ***kwargs*)
Testing code, mainly. Wrapper to cook mnist dataset. Will take as input,

Parameters

- **save_directory** – which directory to save the cooked dataset onto.
- **dataset_parms** – default is the dictionary. Refer to `setup_dataset`
- **preprocess_params** – default is the dictionary. Refer to `setup_dataset`

Notes

This just creates a `data_params` that loads multiple batches without cache. I use this to test the cahcing working on datastream module.

yann.special.datasets.**cook_mnist_normalized**(*verbose=1*, ***kwargs*)
Wrapper to cook mnist dataset. Will take as input,

Parameters

- **save_directory** – which directory to save the cooked dataset onto.
- **dataset_parms** – default is the dictionary. Refer to `setup_dataset`
- **preprocess_params** – default is the dictionary. Refer to `setup_dataset`

Notes

By default, this will create a dataset that is not mean-subtracted.

yann.special.datasets.**cook_mnist_normalized_zero_mean**(*verbose=1*, ***kwargs*)
Wrapper to cook mnist dataset. Will take as input,

Parameters

- **save_directory** – which directory to save the cooked dataset onto.
- **dataset_parms** – default is the dictionary. Refer to `setup_dataset`
- **preprocess_params** – default is the dictionary. Refer to `setup_dataset`

yann.special.datasets.**download_celebA**(*data_dir='celebA'*)
This method downloads celebA dataset into directory _data/`data_dir`.

Parameters **data_dir** – Location to save the data.

class yann.special.datasets.**mix_split_datasets**(*loc*, *verbose=1*, ***kwargs*)
Everything is the same, except, labels are mixed.

class yann.special.datasets.**split_all**(*dataset_init_args*, *save_directory='_datasets'*, *verbose=0*, ***kwargs*)
Inheriting from the setup dataset. The new methods added will include the split.

class yann.special.datasets.**split_continual**(*dataset_init_args*, *save_directory='_datasets'*, *verbose=0*, *n_classes=10*, ***kwargs*)
Inheriting from the setup dataset. This method will produce datasets setup for continual learning systems.

class yann.special.datasets.**split_only_train**(*dataset_init_args*, *save_directory='_datasets'*, *verbose=0*, ***kwargs*)
Inheriting from the split dataset. The new methods added will include the split.

The story behind Yann

I am Ragav Venkatesan, the creator of Yann. I started building convolutional neural networks in early 2015. I began with theano and started following and implementing their tutorials. As I started reading new papers and coding new technologies, I slowly integrated them into what was soon developing into a toolbox. My lab mates at Visual Representaiton and Processing Group also started getting into CNN research and started using my toolbox so I formalized it and hosted it on GitHub. Originally it was a completely unstructured and completely demodularized toolbox and went with the name 'samosa'. The original codebase still exists in older commits on the git. This toolbox still at its core is still the theano tutorials from which it was built.

After considerable effort being put in to make this toolbox modular, and testing it out and after using the toolbox on some of my own research. This toolbox, which began as a pet project is something that I am now proud to share with the rest of DL community with.

This toolbox is also used with the course CSE 591 at ASU in Spring of 2017. With more features being added into the toolbox, I figured I would clean it up, formalize it and write some good documentation so that interested people could use it after. Thus after being rechristened as Yann, this toolbox was born.

Warning: This is a personal toolbox that I wrote as a project for myself. I promise nothing. Although I try to make sure everything is perfectly working. I encourage anyone to use it, contribute to it and write on top of it with that caveat. I still have a lot of code to be written - unittests, optimizers, pre-trained models, dataset ports and for all of these I would really appreciate contributors and help.

Tip: I am working on tutorials and quickstart guides for this toolbox. As such, I try to go in detail in these tutorials, but I am assuming a pre-requisite training and knowledge on CNNs and Neural Neworks. If you are here looking for a tutorial for those and are disappointed with the material here, please read Prof. Yoshua Bengio's book on Deep Learning, or read the examples from theano tutorials. Theano tutorials also will help understand theano which is the backend I used for this toolbox. As the course begins, I will add more tutorials with notes that will make this more useful and interactive.

What is in the toolbox ?

This toolbox started with the beautiful theano tutorials. The code that is here in yann has the following popular features that all deep net toolboxes seem to have. Among many others there are:

- **CNNs with easy architecture management:** Because layers can take origins as inputs, pretty much any architecture that can be drawn on a blackboard can be constructed using this toolbox.
- **Dataset handling capabilities:** Features to setup data as simple matlab files that can be loaded in python and run in batches for training. Alternatively, there is also a wrapper to skdata's dataset interface and future plans for adding the Fuel interface also. As of now, directt ports through `skdata` to cifar10, mnist, extended mnists for Hugo Larochelle , caltech101, caltech256. More to be added soon.
- **Data visualization capabilities:** Visualize the activities of select images (or random) from the train set on each layer after select number of epochs of training. Also view filters after select number of epochs. I find this very effective for my understanding.
- **Techniques from recent publications:** This is where things change a lot and for someone who is getting into deep learning fresh without much theoretical machine learning it would be really helpful. With only a few flags and parameters, one could switch entire optimization methods, including gradient descent, adaGrad, rmsProp add momentums like Polyak Momentum, Nesterov's accelerated gradients etc. One switch converts the whole networks into a max-margin formulation from a softmax formulation. All of these options are plug and play or more like add and cook. Most new methods that are recently published and are added including but not limited to:
  - Dropouts[1]
  - adaGrad[2]
  - Polyak Momentum[3]
  - Nesterov's Accelerated Gradients [4]
  - rmsProp [5]
  - *Adam [8]
  - Maxout and Mixed out Networks [6]

- *FitNets and MentorNets[9,15]
- VGG-19 [10]
- *Inception Module [11]
- Batch Normalization [12]
- ReLU / ELU and other activations supported through theano [13,14]
- Generative Adversarial Networks [16]

Those marked * are not fully tested yet.

References

License

Yann Release: Yann V 1.0rc1

The MIT License

Credits and copyright attributions to other sources:

- First and foremost, this toolbox was develoepd largely by adding functionality to the amazing theano tutorials . The theano tutorial today looks much different from what it was when yann was developed on top of it. Although yann looks completely different now being its own entity, one can still find some philosophy and structure of the theano tutorial code embedded here and there in the toolbox. Therefore enormous credit is due to theano tutorial and theano developers for this toolbox. A lot of developement issues with this toolbox was solved also due to the incredible customer support that was provided by the theano developers at their google groups forum and the issues section of the theano github page. Copyright (c) 2008–2016, Theano Development Team All rights reserved.
- The dropouts part of the code was adapted from Misha Denil's Dropout code. The code at the moment does not look at all like the code that Misa Denil originally had, but it was developed from his code and therefore requires that due credit be assigned and acknowledgement provided. Refer to license here . Including his copyright notice here. Copyright (C) 2012 Misha Denil
- Ofcourse, this toolbox contains code from numpy, Contains code from NumPy, Copyright (c) 2005-2011, NumPy Developers. All rights reserved.

- Lasagne played a signnificant role in the developement of yann. Part of the intention in begining to develop yann was that Lasagne was getting too large for the authors to keep up. Lasagne therefore insipired some of the aspects of this toolbox. One of the contributors who submitted the rotate layer, used code re-distributed from Lasagne toolbox as well. So, in honoring their license, Copyright (c) 2014-2016 Lasagne contributors.
- Credit is also due to developers of skdata, progressbar, networkx and developers of all the other dependencies that helping in the creating of this toolbox. Go team OpenSource!

Indices of functions and modules

- genindex
- modindex

Python Module Index

p

y

Index

E

F

G

I

L

M

N

O

P

R

S

T

U

V

Y