pydnn Documentation

Release 0.0

Isaac Kriegman

April 02, 2015

Contents

CHAPTER 1

pydnn: High performance GPU neural network library for deep learning in Python

pydnn is a deep neural network library written in Python using Theano (symbolic math and optimizing compiler package). It was written for Kaggle's National Data Science Bowl competition in March 2015, where it produced an entry finishing in the top 6%. Continued development is planned, including support for even more of the most important deep learning techniques (RNNs...)

Contents

1.1 Design Goals

- **Simplicity** Wherever possible simplify code to make it a clear expression of underlying deep learning algorithms. Minimize cognitive overhead, so that it is easy for someone who has completed the deeplearning.net tutorials to pickup this library as a next step and easily start learning about, using, and coding more advanced techniques.
- **Completeness** Include all the important and popular techniques for effective deep learning and **not** techniques with more marginal or ambiguous benefit.
- **Ease of use** Make preparing a dataset, building a model and training a deep network only a few lines of code; enable users to work with NumPy rather than Theano.
- **Performance** Should be roughly on par with other Theano neural net libraries so that pydnn is a viable choice for computationally intensive deep learning.

1.2 Features

- High performance GPU training (courtesy of Theano)
- Quick start tools to instantly get started training on inexpensive Amazon EC2 GPU instances.
- **Implementations of important new techniques recently reported in the literature:**
  - Batch Normalization
  - Parametric ReLU activation function,
  - Adam optimization
  - AdaDelta optimization
  - etc.
- **Implementations of standard deep learning techniques:**
  - Stochastic Gradient Descent with Momentum
  - Dropout
  - convolutions with max-pooling using overlapping windows
  - ReLU/Tanh/sigmoid activation functions
  - etc.

1.3 Documentation

http://pydnn.readthedocs.org/en/latest/index.html

1.4 Installation

pip install pydnn

1.5 Home Page

https://github.com/zackriegman/pydnn

1.6 Usage

First download and unzip raw image data from somewhere (e.g. Kaggle). Then:

```
import pydnn
import numpy as np
rng = np.random.RandomState(e.rng_seed)

# build data, split into training/validation sets, preprocess
train_dir = 'home\ubuntu\train'
data = pydnn.data.DirectoryLabeledImageSet(train_dir).build()
data = pydnn.preprocess.split_training_data(data, 64, 80, 15, 5)
```

```
resizer = pydnn.preprocess.StretchResizer()
pre = pydnn.preprocess.Rotator360(data, (64, 64), resizer, rng)

# build the neural network
net = pydnn.nn.NN(pre, 'images', 121, 64, rng, pydnn.nn.relu)
net.add_convolution(72, (7, 7), (2, 2))
net.add_dropout()
net.add_convolution(128, (5, 5), (2, 2))
net.add_dropout()
net.add_convolution(128, (3, 3), (2, 2))
net.add_dropout()
net.add_hidden(3072)
net.add_dropout()
net.add_hidden(3072)
net.add_dropout()
net.add_logistic()

# train the network
lr = pydnn.nn.Adam(learning_rate=pydnn.nn.LearningRateDecay(
            learning_rate=0.006,
            decay=.1))
net.train(lr)
```

From raw data to trained network (including specifying network architecture) in 25 lines of code.

1.7 Short Term Goals

- Implement popular RNN techniques.
- Integrate with Amazon EC2 clustering software (such as StarCluster).
- Integrate with hyper-parameter optimization frameworks (such as Spearmint and hyperopt).

1.8 Authors

Isaac Kriegman

CHAPTER 2

pydnn.neuralnet module

2.1 Overview

`NN` is the workhorse of pydnn. Using an instance of `NN` the user defines the network, trains the network and uses the network for inference. `NN` takes care of the bookkeeping and wires the layers together, calculating any intermediate configuration necessary for doing so without user input. See the *section on NN* for more details.

Learning rules define how the network updates weights based on the gradients calculated during training. Learning rules are passed to `NN` objects when calling `NN.train()` to train the network. `Momentum` and `Adam` are good default choices. See *Learning Rules (Optimization Methods)* for more details.

All the learning rules defined in this package depend in part on a global learning rate that effects how all parameters are updated on training passes. It is frequently beneficial to anneal the learning rate over the course of training and different approaches to annealing can result in substantially different convergence losses and times. Different approaches to annealing can be achieved by using one of the various learning rate annealing objects which are passed to `LearningRule` objects during instantiation. `LearningRateDecay` is a good default choice. See *Learning Rate Annealing* for more details.

A variety of activation functions, or nonlinearities, can be applied to layers. `relu()` is the most common, however `PReLULayer` has recently been reported to achieve state of the art results. See *Activation Functions (Nonlinearities)* for more details.

Finally there are a few utilities for saving and reloading trained networks and for estimating the size and training time for networks before training. See *Utility Functions* for more details.

Contents

2.2 The main class: NN

class pydnn.neuralnet.**NN**(*preprocessor*, *channel*, *num_classes*, *batch_size*, *rng*, *activation=<function relu at 0x7f4c2cc9a410>*, *name='net'*, *output_dir=''*)

a neural network to which you can add layers and subsequently train on data

Parameters

- **preprocessor** – a preprocessor for the data provides all the training, validation and test data to NN during training
- **channel** (*string*) – the initial channel to request from the preprocessor for the main layer pathway
- **num_classes** (*int*) – the number of classes
- **batch_size** (*int*) – the number of observations in a batch
- **rng** – random number generator
- **activation** – the default activation function to be used when no activation function is explicitly provided
- **name** (*string*) – a name to use as a stem for saving network parameters during training
- **output_dir** (*string*) – the directory in which to save network parameters during training

Networks are constructed by calling the add_*() methods in sequence to add processing layers. For example:

```
net.add_convolution(72, (7, 7), (2, 2))
net.add_dropout()
net.add_convolution(128, (5, 5), (2, 2))
net.add_dropout()
net.add_convolution(128, (3, 3), (2, 2))
net.add_dropout()
net.add_hidden(3072)
net.add_dropout()
net.add_hidden(3072)
net.add_dropout()
net.add_logistic()
```

The above creates a network with three convolutional layers with 72, 128 and 128 filter maps respectively, two hidden layers, each with 3072 units, dropout with a rate of 0.5 between each main layer and batch normalization (by default on each main layer).

There are a few convenience add_*() methods which are just combinations of other add methods: add_convolution(), add_mlp() and add_hidden().

There are a few methods for creating different processing pathways that can split from and rejoin the main network. For example:

```
net.add_convolution(72, (7, 7), (2, 2))
net.add_dropout()
net.add_convolution(128, (5, 5), (2, 2))
net.add_dropout()
net.merge_data_channel('shapes')
net.add_hidden(3072)
net.add_dropout()
net.add_logistic()
```

Here a new data channel called *'shapes'* was merged after the convolution. *'shapes'* is a channel provided by the preprocessor with the original image sizes. (This can be useful where image sizes vary in meaningful

ways; since that information is lost when uniformly resizing images to be fed into the neural network, it can be recovered by feeding in the size information separately after convolutions.) In addition to simply merging a new data channel, it is also possible to split off a new pathway, apply transformations to it, and merge it back to the main pathway with `split_pathways()`, `merge_pathways()`, `new_pathway()`.

Once a neural network architecture has been built up, the network can be trained with `train()`. After training, inference can be done with `predict()`, and confusion matrices can be generated with `get_confusion_matrices()` to examine the kinds of errors the network is making.

new_pathway(*channel*)
Creates a new pathway starting from a data channel. (After adding layers specific to this pathway, if any, the new pathway must subsequently be merged with main pathway using `merge_pathways()`.)

Parameters **channel** (*string*) – name of the channel as output from preprocessor

Returns `NN` to which layers can be separately added

split_pathways(*num=None*)
Splits pathways off from the `NN` object. Split pathways can have different sequences of layers added and then be remerged using `merge_pathways()`.

Parameters **num** (*int*) – number of new pathways to split off from the original pathway. If `num` is `None` then split just one new pathway.

Returns If `num` is not `None`, returns a list of the new pathways (not including the original pathway); otherwise returns a single new pathway.

NOTE: NN.params list is not copied when splitting pathways, meaning that when any pathway adds a layer, the params for that layer are added to the params of all pathways (since there is only one params list). Normally, this will not cause a problem, however, if a pathway is split but not merged back in to the trunk (and far as I can think of there is no reason to do this) then updates will be generated for parameters that are not in the computation graph and theano will probably throw and exception. If we consider it an error to split pathways without eventually remerging them, then this is not a problem.

merge_pathways(*pathways*)
Merge pathways.

Parameters **pathways** – pathways to merge. *pathways* can be a single pathway or a list of pathways

merge_data_channel(*channel*)
Creates a new pathway for processing *channel* data and merges it without adding any pathway specific layers.

Parameters **channel** (*string*) – name of the channel as output from preprocessor

add_conv_pool(*num_filters*, *filter_shape*, *pool_shape*, *pool_stride=None*, *weight_init=None*, *use_bias=True*)
Adds a convolution and max pooling layer to the network (without a nonlinearity or batch normalization; if those are desired they can be added separately, or the convenience method `add_convolution()` can be used).

Parameters

- **num_filters** (*int*) – number of filter maps to create
- **filter_shape** (*tuple*) – two dimensional shape of filters
- **pool_shape** (*tuple*) – two dimensional shape of pools
- **pool_stride** (*tuple*) – distance between pool starting points; if this is less than *pool_shape* then pools will be overlapping

- **weight_init** – activation function that will be applied to for the purposes of initializing weights (this method will not apply the activation function; it must be added separately as a layer). One of `relu()`, `tanh()`, `sigmoid()`, or `prelu()`
- **use_bias** (*bool*) – `True` for bias, `False` for no bias. No bias should be used when batch normalization layer will be processing the output of this layer (e.g. when `add_batch_normalization()` is called next).

add_convolution(*num_filters*, *filter_shape*, *pool_shape*, *pool_stride=None*, *activation=None*, *batch_normalize=True*)

Adds a convolution, pooling layer and nonlinearity to the network (with the option of a batch normalization layer).

Parameters

- **num_filters** (*int*) – number of filter maps to create
- **filter_shape** (*tuple*) – two dimensional shape of filters
- **pool_shape** (*tuple*) – two dimensional shape of pools
- **pool_stride** (*tuple*) – distance between pool starting points; if this is less than *pool_shape* then pools will be overlapping
- **activation** – activation function to be applied to pool output. (One of `relu()`, `tanh()`, `sigmoid()`, or `prelu()`)
- **batch_normalize** (*bool*) – `True` for batch normalization, `False` for no batch normalization.

add_fully_connected(*num_units*, *weight_init*, *use_bias*)

Add a layer that does a matrix multiply and addition of biases. (No nonlinearity is applied in this layer because when batch normalization is applied it must come between the matrix multiply and the nonlinearity. A nonlinearity can be applied either by using the `add_hidden()` convenience method instead of this one or by subsequently calling `add_nonlinearity()`.)

Parameters

- **num_units** (*int*) – number of neurons in the fully connected layer
- **weight_init** – activation function that will be applied after the fully connected layer (used to determine a weight initialization scheme–one of `relu()`, `tanh()`, `sigmoid()`, or `prelu()`)
- **use_bias** (*bool*) – `True` to use bias; `False` not to. (When using batch normalization, bias is redundant and thus should not be used.)

add_hidden(*num_units*, *activation=None*, *batch_normalize=True*)

Add a hidden layer consisting of a fully connected layer, a nonlinearity layer, and optionally a batch normalization layer. (The equivalent of calling `add_fully_connected()`, `add_batch_normalization()`, and `add_nonlinearity()` in sequence.)

Parameters

- **num_units** (*int*) – number of neurons in the hidden layer
- **activation** – activation function to be applied
- **batch_normalize** (*bool*) – `True` for batch normalization, `False` for no batch normalization.

add_nonlinearity(*nonlinearity*)

Add a layer which applies a nonlinearity to its inputs.

Parameters **nonlinearity** – the activation function to be applied. (One of `relu()`, `tanh()`, `sigmoid()`, or `prelu()`)

add_dropout(*rate=0.5*)
Add a dropout layer.

See the dropout paper.

Randomly mask inputs with zeros with frequency `rate` while training, and scales inputs by `1.0 - rate` when not training so that aggregate signal sent to next layer will be roughly the same during training and inference.

Parameters **rate** (*float*) – rate at which to randomly zero out inputs

add_batch_normalization(*epsilon=1e-06*)
Add a batch normalization layer

See the batch normalization paper

add_logistic()
Add a logistic classifier (should be the final layer).

add_mlp(*num_hidden_units*, *activation=None*)
A convenience function for adding a hidden layer and logistic regression layer at the same time. (Mostly here to mirror deeplearning.net tutorial.)

Parameters

- **num_hidden_units** (*int*) – number of hidden units
- **activation** – activation function to be applied to hidden layer output. (One of `relu()`, `tanh()`, `sigmoid()`, or `prelu()`)

train(*updater*, *epochs=200*, *final_epochs=0*, *l1_reg=0*, *l2_reg=0*)
Train the model

Parameters

- **updater** – the learning rule; one of `StochasticGradientDescent`, `Adam`, `AdaDelta`, or `Momentum`
- **epochs** (*int*) – the number of epochs to train for
- **final_epochs** (*int*) – the number of final epochs to train for. (Final epochs are epochs where the validation and test data are folded into the training data for a little boost in the size of the dataset.)
- **l1_reg** (*float*) – l1 regularization penalty
- **l2_reg** (*float*) – l2 regularization penalty

predict(*data*)
Predict classes for input data.

Parameters **data** (*ndarray*) – data to be processed in order to make prediction

Returns (list of predicted class indexes for each inference observation, list of assessed probabilities for each class possibility for each inference observation)

make_confusion_matrix(*data*, *classes*, *files*)
Make a confusion matrix given input data and correct class designations

Parameters

- **data** (*ndarray*) – the data for which classes are predicted

- **classes** (*ndarray*) – the correct classes to be compared with the predictions
- **files** – an id/index for each observation to facilitate connecting them back up to filenames

Returns (confusion matrix, list of mistakes (file_index, actual, pred))

get_confusion_matrices()
Run `make_confusion_matrix()` on training, validation and test data and return list of results.

Returns list of confusion matrices for training, validation, and test data

2.3 Learning Rules (Optimization Methods)

class pydnn.neuralnet.**LearningRule**(*learning_rate*)
Base class for learning rules: `StochasticGradientDescent`, `Adam`, `AdaDelta`, `Momentum`.

Parameters learning_rate – either a float (if using a constant learning rate) or a `LearningRateAdjuster` (if using a learning rate that is adjusted during training)

class pydnn.neuralnet.**StochasticGradientDescent**(*learning_rate*)
Learn by stochastic gradient descent

class pydnn.neuralnet.**Momentum**(*initial_momentum*, *max_momentum*, *learning_rate*)
Learn by SGD with momentum.

Parameters

- **initial_momentum** (*float*) – trainings starts with this momentum
- **max_momentum** (*float*) – momentum is gradually increased until it reaches *max_momentum*

class pydnn.neuralnet.**Adam**(*learning_rate*, *b1=0.9*, *b2=0.999*, *e=1e-08*, *lmbda=0.99999999*)
Learn by the Adam optimization method

Parameters are as specified in the paper above.

class pydnn.neuralnet.**AdaDelta**(*rho*, *epsilon*, *learning_rate*)
Learn by the AdaDelta optimization method

Parameters are as specified in the paper above.

2.4 Learning Rate Annealing

class pydnn.neuralnet.**LearningRateAdjuster**(*initial_learn_rate*)
Base class for learning rate annealing: `LearningRateDecay`, `LearningRateSchedule`, and `WackyLearningRateAnnealer`.

Parameters initial_learn_rate (*float*) – the learning rate to start with on the first epoch

class pydnn.neuralnet.**LearningRateDecay**(*learning_rate*, *decay*, *min_learning_rate=None*)
Decreases learning rate after each epoch according to formula: `new_rate = initial_rate / (1 + epoch * decay)`

Parameters

- **learning_rate** (*float*) – the initial learning rate
- **decay** (*float*) – the decay factor

- **min_learning_rate** (*float*) – the smallest `learning_rate` to which decay is applied; when `learning_rate` reaches `min_learning_rate` decay stops.

class `pydnn.neuralnet.`**`LearningRateSchedule`**(*schedule*)

Sets the learning rate according to the given schedule.

Parameters **schedule** (*tuple*) – list of pairs of epoch number and new learning rate. For example, `((0, .1), (200, .01), (300, .001))` starts with a learning rate of .1, changes to a learning rate of .01 at epoch 200, and .001 at epoch 300.

class `pydnn.neuralnet.`**`WackyLearningRateAnnealer`**(*learning_rate*, *min_learning_rate*, *patience=40*, *train_improvement_threshold=0.995*, *valid_improvement_threshold=0.99995*, *reset_on_decay=None*)

Decreases learning rate by factor of 10 after patience is depleted. Patience can be replenished by sufficient improvement in either training or validation loss. Parameters of the network can optionally be reset to the parameters corresponding to the best training loss or to the best validation loss.

Parameters

- **learning_rate** (*float*) – the initial learning rate
- **min_learning_rate** (*float*) – training stops upon reaching the min_learning_rate
- **patience** (*int*) – the number of epochs to train without sufficient improvement in training or validation loss before dropping the learning rate
- **train_improvement_threshold** (*float*) – how much training loss must improve over previous best training loss to trigger a reset of patience (if `training_loss < best_training_loss * train_improvement_threshold` then patience is reset)
- **valid_improvement_threshold** (*float*) – how much validation loss must improve over previous best validation loss to trigger a reset of patience (if `validation_loss < best_validation_loss * valid_improvement_threshold` then patience is reset)
- **reset_on_decay** (*string*) – one of *'training'*, *'validation'* or *None*; if *'training'* or *'validation'* then on learning rate decay network will be reset to the parameter values that correspond to the best training or validation scores.

2.5 Activation Functions (Nonlinearities)

`pydnn.neuralnet.`**`relu`**(*x*)

Used to create a rectified linear activation layer. The user does not use this directly, but instead passes the function as the `activation` or `weight_init` argument to `NN` either when creating it or adding certain kinds of layers.

Parameters **x** (*float*) – input to the rectified linear unit

Returns 0 if x < 0, otherwise x

Return type float

`pydnn.neuralnet.`**`prelu`**()

Used to create a parametric rectified linear activation layer.

Parametric Rectified Linear Units: http://arxiv.org/pdf/1502.01852.pdf.

The user does not use this directly, but instead passes the function as the `activation` or `weight_init` argument to `NN` either when creating it or adding certain kinds of layers. (This is just a dummy function provided for API consistency with `relu()`, `tanh()` and `sigmoid()`. Unlike those functions it doesn't actually do anything, but merely signals `add_nonlinearity()` to add a parametric rectified nonlinearity)

Note: Don't use l1/l2 regularization with PReLU. From the paper: "It is worth noticing that we do not use weight decay (l2 regularization) when updating a_i. A weight decay tends to push a_i to zero, and thus biases PReLU toward ReLU."

`pydnn.neuralnet.`**`sigmoid`**(*x*)

Used symbolic logistic activation layer. The user does not use this directly, but instead passes the function as the `activation` or `weight_init` argument to `NN` either when creating it or adding certain kinds of layers.

Parameters **x** (*float*) – input to the sigmoid unit

Returns symbolic logistic function of x

Return type float

`pydnn.neuralnet.`**`tanh`**(*x*)

Used to create a hyperbolic tangent activation layer. The user does not use this directly, but instead passes the function as the `activation` or `weight_init` argument to `NN` either when creating it or adding certain kinds of layers.

Parameters **x** (*float*) – input to the hyperbolic tangent unit

Returns symbolic hyperbolic tangent function of x

Return type float

2.6 Utility Functions

`pydnn.neuralnet.`**`save`**(*nn*, *filename=None*)

save a `NN` object to file

Parameters

- **nn** – the `NN` to be saved
- **filename** (*string*) – the path/filename to save to

Returns the filename

`pydnn.neuralnet.`**`load`**(*filename*)

load a `NN` object from file

Parameters **filename** (*string*) – the path/filename to load from

Returns the `NN` object loaded

`pydnn.neuralnet.`**`net_size`**(*root*, *layers*)

A simple utility to calculate the computational size of the network and give a very rough estimate of how long it will take to train. (Ignoring the cost of the activation function, batch_normalization, prelu parameters, and a zillion other things.)

Parameters

- **root** (*tuple*) – image shape (channels, height, width)
- **layers** (*tuple*) – list of layers where each layer is either a conv layer specification or a fully connected layer specification. E.g.: ('conv', {'filter': (192, 3, 3), 'pool': (3, 3), 'pool_stride': (2, 2)}), or ('full', {'num': 3072})

CHAPTER 3

pydnn.preprocess module

3.1 Overview

Most of the code in this module is currently pretty specific to processing images like those in Kaggle's plankton competition. Those images were unique in that they (1) were presented with a uniform background, (2) they varied in size in a way that provided meaningful information about the subject, and (3) they were mostly randomly oriented. These features have to do with real world constraints on the way that marine biologist collect the images, and are obviously quite different from popular datasets like ImageNet, MNIST, etc. As I (or others) use pydnn in a greater variety of machine learning contexts a variety of preprocessing approaches can be maintained here.

Contents

3.2 Training Set

pydnn.preprocess.**split_training_data**(*data*, *batch_size*, *train_per*, *valid_per*, *test_per*)

Split training data into training set, validation set and test sets. If split results in incomplete batches this function will allocate observations from incomplete batches in validation and test sets to the training set to attempt to make a complete batch. This function also reports on the split and whether there were observations that did not fit evenly into any batch. (Currently `NN` assumes a validation and test set, so if allocating 100% of data to training set, `split_training_data()` will duplicate the first batch of the training set for the validation and test sets so `NN` does not fail. In that case, obviously the validation and test loss will be meaningless.)

Parameters

- **data** (*tuple*) – all the data, including x and y values
- **batch_size** (*int*) – number of observations that will be in each batch
- **train_per** (*float*) – percentage of data to put in training set
- **valid_per** (*float*) – percentage of data to put in validation set
- **test_per** (*float*) – percentage of data to put in test set

Returns a list containing the training, validation and test sets, each of which is a list containing each variable (in the case where x data is a single image that will be a list of images and a list of y classes, but there can also be more than one x variable).

3.3 Preprocessors

Preprocessors take care of online augmentation, shuffling, zero centering and normalizing, resizing, and other related transformations of the traning data. Because the plankton images could be in any orientation, to achieve good performance it was important to augment the data with many rotations of the training set so the network could learn to recognize images in different orientations. Initially I experimented with 90 degree rotations and a flip, however I found that unconstrained degree rotations (`Rotator360` and `Rotator360PlusGeometry`) performed better. Another approach that I experimented with was rotating and flipping all images into a canonicalized orientation based on their shape and size (`Canonicalizer`), which significantly improves early training progress, but shortly thereafter falls behind a 360 degree rotation approach.

Another thing that these preprocessors do is add additional data channels. For instance, since, in the case of the plankton dataset, the size of the images carries important information (because image size was related to the size of the organism) it was useful to add a data channel with the original image size (`Rotator360`), because that information is lost when uniformly resizing images to be fed into the neural network. Another approach, instead of the original image size, was to create a channel with the size of the largest contiguous image shape, and it's rotation in comparison to it's canonicalized rotation (`Rotator360PlusGeometry`).

class pydnn.preprocess.**Rotator360**(*data*, *image_shape*, *resizer*, *rng*, *dtype='float32'*)

Rotates training set images randomly.

(Also zero centers by pixel, normalizes, shuffles, resizes, etc.)

Parameters

- **data** – x and y data
- **image_shape** – the target image shape
- **resizer** – the resizer to use when uniformly resizing images
- **rng** – random number generator
- **dtype** (*string*) – the datatype to output

class pydnn.preprocess.**Rotator360PlusGeometry**(*data*, *image_shape*, *resizer*, *rng*, *dtype*)

Rotates training set images randomly, but also generates additional geometric data about the size and orientation of the organism in the image.

(Also zero centers by pixel, normalizes, shuffles, resizes, etc.)

Parameters

- **data** – x and y data
- **image_shape** – the target image shape
- **resizer** – the resizer to use when uniformly resizing images
- **rng** – random number generator
- **dtype** (*string*) – the datatype to output

class pydnn.preprocess.**Canonicalizer**(*data*, *image_shape*, *resizer*, *rng*, *dtype*)

Rotates and flips all images into a canonicalized form. Using a statistical measure of object height rotates each image to minimize height. (Can also either (1) flip images so aggregate pixel intensity is highest in one corner, or (2) generate random flips of training images.)

(Also zero centers by pixel, normalizes, shuffles, resizes, etc.)

Parameters

- **data** – x and y data
- **image_shape** – the target image shape
- **resizer** – the resizer to use when uniformly resizing images
- **rng** – random number generator
- **dtype** (*string*) – the datatype to output

3.4 Resizing

Users do not use the resizers directly but pass them to a *preprocessor* to control how the preprocessor resizes images.

`pydnn.preprocess.`**`Resizer`**`()`
Base class for `StretchResizer`, `ContiguousBoxPreserveAspectRatioResizer`, `ContiguousBoxStretchResizer`, `ThresholdBoxPreserveAspectRatioResizer`, `ThresholdBoxStretchResizer`, `PreserveAspectRatioResizer`, and `StochasticStretchResizer`

Some resizers may want to resize training images differently from validation or testing images so this class gives them the option of doing so.

`pydnn.preprocess.`**`StretchResizer`**`()`
Stretches the images to a uniform shape ignoring aspect ratio

`pydnn.preprocess.`**`ContiguousBoxPreserveAspectRatioResizer`**`(`*threshold*`)`
First crops the images around the largest contiguous region, then stretches them to a uniform size preserving aspect ratio.

`pydnn.preprocess.`**`ContiguousBoxStretchResizer`**`(`*threshold*`)`
First crops the images around the largest contiguous region, then stretches them to a uniform ignoring aspect ratio.

`pydnn.preprocess.`**`ThresholdBoxPreserveAspectRatioResizer`**`(`*threshold*`)`
First crops the images, throwing away outside space without pixels that exceed a given threshold, then stretches them to a uniform size preserving aspect ratio.

`pydnn.preprocess.`**`ThresholdBoxStretchResizer`**`(`*threshold*`)`
First crops the images, throwing away outside space without pixels that exceed a given threshold, then stretches them to a uniform ignoring aspect ratio.

`pydnn.preprocess.`**`PreserveAspectRatioResizer`**`()`
Stretches images to a uniform size preserving aspect ratio.

`pydnn.preprocess.`**`StochasticStretchResizer`**`(`*rng*, *rand_range*`)`
Stretches images to a uniform size ignoring aspect ratio.

Randomly varies how much training images are stretched.

CHAPTER 4

pydnn.data module

class pydnn.data.**DirectoryLabeledImageSet**(*base_dir*, *dtype='float32'*)

Bases: object

Builds training data from a directory where each subdirectory is the name of a class, and contains all the examples of images that class.

Parameters

- **base_dir** (*string*) – the directory containing the class directories
- **dtype** (*string*) – the data type to use for the ndarray containing the labels

build(*num_images=None*)

get_files(*num_files=None*)

get_labels()

get_random_file(*rng*)

class pydnn.data.**UnlabeledImageSet**(*base_dir*)

Bases: object

Builds an inference set from a directory containing unlabeled images.

Parameters **base_dir** (*string*) – the directory containing the images

build(*start*, *stop*)

get_files()

CHAPTER 5

pydnn.aws_util module

pydnn.aws_util.**create_security_group**(*conn*)

pydnn.aws_util.**get_image_by_name**(*conn*, *name*)

pydnn.aws_util.**get_image_id_by_name**(*conn*, *name*)

pydnn.aws_util.**get_login_exec**(*instance_ip*)

pydnn.aws_util.**get_recent_gpu_price**(*conn*)

pydnn.aws_util.**get_running_instances**(*conn*, *name=None*)

pydnn.aws_util.**get_sftp_exec**(*instance_ip*)

pydnn.aws_util.**get_unique_instance**(*conn*, *name*)

pydnn.aws_util.**handle_command_line**()

pydnn.aws_util.**launch_instance**(*conn*, *name*, *image_id*)

pydnn.aws_util.**list_amis**(*conn*)

pydnn.aws_util.**list_instances**(*conn*)

pydnn.aws_util.**print_instances**(*instances*)

pydnn.aws_util.**save_spot_instance**(*conn*, *name*, *image_name*, *terminate=True*)

pydnn.aws_util.**sftp**(*conn*, *name*)

pydnn.aws_util.**ssh**(*conn*, *name*)

pydnn.aws_util.**start_spot_instance**(*conn*, *name*, *image_id=None*)

pydnn.aws_util.**stop_all_spot_instances_without_saving**(*conn*)

pydnn.aws_util.**stop_spot_instance_without_saving**(*conn*, *instance_id*)

CHAPTER 6

pydnn.img_util module

pydnn.img_util.**dimensions_to_fit_images**(*length*)

pydnn.img_util.**show_image_tiles**(*images*, *canvas_dims=None*)

pydnn.img_util.**show_images**(*images*, *titles=None*, *canvas_dims=None*)

pydnn.img_util.**show_images_as_tiles**(*images*, *size*, *canvas_dims=None*)

pydnn.img_util.**test_show_images**()

CHAPTER 7

pydnn.tools module

class pydnn.tools.**Dot**(*skip=0*)

Bases: object

dot(*string=None*)

stop()

pydnn.tools.**H**(*num*)

pydnn.tools.**default**(*variable*, *dfault*)

pydnn.tools.**get_files**(*directory*, *rel=False*, *cache=False*)

pydnn.tools.**get_sub_dirs**(*directory*, *rel=False*, *cache=False*)

pydnn.tools.**h**(*num*)

pydnn.tools.**hum**(*num*)

pydnn.tools.**human**(*num*)

pydnn.tools.**image_tile**(*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).

Parameters

- **X** (*a 2-D ndarray or a tuple of 4 channels, elements of which can be 2-D ndarrays or None;*) – a 2-D array in which every row is a flattened image.
- **img_shape** (*tuple; (height, width)*) – the original shape of each image
- **tile_shape** (*tuple; (rows, cols)*) – the number of images to tile (rows, cols)
- **output_pixel_vals** – if output should be pixel values (i.e. int8 values) or floats
- **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*.)

Return type a 2-d array with same dtype as X.

pydnn.tools.**load_config**(*environ_variable*, *module_file*, *default_config*)

pydnn.tools.**now**()

pydnn.tools.**num_abbrev**(*num*, *abbrev*, *sep*)

pydnn.tools.**raise_exception**(*x*)

pydnn.tools.**save_output**(*filename*, *func*, **args*, ***kw*)

pydnn.tools.**scale_to_unit_interval**(*ndar*, *eps=1e-08*)
Scales all values in the ndarray ndar to be between 0 and 1

pydnn.tools.**send_email**(*from_addr*, *to_addr*, *username*, *password*, *smtp*, *subject=''*, *body=''*)

pydnn.tools.**tile_2d_images**(*images*, *canvas_shape*)

pydnn.tools.**time_once**(*method*)

CHAPTER 8

examples.plankton package

examples.plankton.plankton.**analyze_confusion_matrix**(*matrix_file*)

examples.plankton.plankton.**average_submissions**(*in_files*, *weights=None*)

examples.plankton.plankton.**generate_submission_file**(*net*, *name*, *num=None*)

examples.plankton.plankton.**load_net_and_generate_submission_file**(*net_name*, *submission_name*)

examples.plankton.plankton.**make_confusion_matrix_from_saved_network**(*e*)

examples.plankton.plankton.**run_experiment**(*e*)

examples.plankton.plankton.**show_mistakes**(*mistakes_file*)

examples.plankton.plankton.**write_confusion_matrices_to_csv_files**(*experiment*, *num_images*, *matrices*)

examples.plankton.plankton.**write_submission_csv_file**(*file_name*, *probs*, *image_file_names*)

- *genindex*
- *modindex*
- *search*

e

p

A

B

C

D

E

G

H

I

L