# nolearn Documentation

***Release 0.6***

**Daniel Nouri**

September 06, 2016

Contents

This package contains a number of utility modules that are helpful with machine learning tasks. Most of the modules work together with scikit-learn, others are more generally useful.

nolearn's source is hosted on Github. Releases can be downloaded on PyPI.

CHAPTER 1

# Installation

We recommend using virtualenv to install nolearn.

To install the latest version of nolearn from Git using pip, run the following commands:

```
pip install -r https://raw.githubusercontent.com/dnouri/nolearn/master/requirements.txt
pip install git+https://github.com/dnouri/nolearn.git@master#egg=nolearn==0.7.git
```

To instead install the release from PyPI (which is somewhat old at this point), do:

```
pip install nolearn
```

CHAPTER 2

# Modules

## 2.1 nolearn.cache

This module contains a decorator *cached()* that can be used to cache the results of any Python functions to disk.

This is useful when you have functions that take a long time to compute their value, and you want to cache the results of those functions between runs.

Python's `pickle` is used to serialize data. All cache files go into the *cache/* directory inside your working directory.

*@cached* uses a cache key function to find out if it has the value for some given function arguments cached on disk. The way it calculates that cache key by default is to simply use the string representation of all arguments passed into the function. Thus, the default cache key function looks like this:

```
def default_cache_key(*args, **kwargs):
    return str(args) + str(sorted(kwargs.items()))
```

Here is an example use of the *cached()* decorator:

```
import math
@cached()
def fac(x):
    print 'called!'
    return math.factorial(x)

fac(20)
called!
2432902008176640000
fac(20)
2432902008176640000
```

Often you will want to use a more intelligent cache key, one that takes more things into account. Here's an example cache key function for a cache decorator used with a *transform* method of a scikit-learn `BaseEstimator`:

```
>>> def transform_cache_key(self, X):
...     return ','.join([
...         str(X[:20]),
...         str(X[-20:]),
...         str(X.shape),
...         str(sorted(self.get_params().items())),
...         ])
```

This function puts the first and the last twenty rows of the matrix *X* into the cache key. On top of that, it adds the shape of the matrix *X.shape* along with the items in *self.get_params*, which with a scikit-learn `BaseEstimator` class is

the dictionary of model parameters. This makes sure that even though the input matrix is the same, it will still calculate the value again if the value of *self.get_params()* is different.

Your estimator class can then use the decorator like so:

```
class MyEstimator(BaseEstimator):
    @cached(transform_cache_key)
    def transform(self, X):
        # ...
```

nolearn.cache.**cached**(*cache_key=<function default_cache_key>*, *cache_path=None*)

## 2.2 nolearn.dbn

**Warning:** The nolearn.dbn module is no longer supported. Take a look at *nolearn.lasagne* for a more modern neural net toolkit.

### 2.2.1 API

**class** nolearn.dbn.**DBN**(*layer_sizes=None*, *scales=0.05*, *fan_outs=None*, *output_act_funct=None*, *real_valued_vis=True*, *use_re_lu=True*, *uniforms=False*, *learn_rates=0.1*, *learn_rate_decays=1.0*, *learn_rate_minimums=0.0*, *momentum=0.9*, *l2_costs=0.0001*, *dropouts=0*, *nesterov=True*, *nest_compare=True*, *rms_lims=None*, *learn_rates_pretrain=None*, *momentum_pretrain=None*, *l2_costs_pretrain=None*, *nest_compare_pretrain=None*, *epochs=10*, *epochs_pretrain=0*, *loss_funct=None*, *minibatch_size=64*, *minibatches_per_epoch=None*, *pretrain_callback=None*, *fine_tune_callback=None*, *random_state=None*, *verbose=0*)

A scikit-learn estimator based on George Dahl's DBN implementation *gdbn*.

**__init__**(*layer_sizes=None*, *scales=0.05*, *fan_outs=None*, *output_act_funct=None*, *real_valued_vis=True*, *use_re_lu=True*, *uniforms=False*, *learn_rates=0.1*, *learn_rate_decays=1.0*, *learn_rate_minimums=0.0*, *momentum=0.9*, *l2_costs=0.0001*, *dropouts=0*, *nesterov=True*, *nest_compare=True*, *rms_lims=None*, *learn_rates_pretrain=None*, *momentum_pretrain=None*, *l2_costs_pretrain=None*, *nest_compare_pretrain=None*, *epochs=10*, *epochs_pretrain=0*, *loss_funct=None*, *minibatch_size=64*, *minibatches_per_epoch=None*, *pretrain_callback=None*, *fine_tune_callback=None*, *random_state=None*, *verbose=0*)

Many parameters such as *learn_rates*, *dropouts* etc. will also accept a single value, in which case that value will be used for all layers. To control the value per layer, pass a list of values instead; see examples below.

Parameters ending with *_pretrain* may be provided to override the given parameter for pretraining. Consider an example where you want the pre-training to use a lower learning rate than the fine tuning (the backprop), then you'd maybe pass something like:

```
DBN([783, 300, 10], learn_rates=0.1, learn_rates_pretrain=0.005)
```

If you don't pass the *learn_rates_pretrain* parameter, the value of *learn_rates* will be used for both pre-training and fine tuning. (Which seems to not work very well.)

**Parameters**

- **layer_sizes** – A list of integers of the form `[n_vis_units, n_hid_units1, n_hid_units2, ..., n_out_units]`.

An example: `[784, 300, 10]`

The number of units in the input layer and the output layer will be set automatically if you set them to -1. Thus, the above example is equivalent to `[-1, 300, -1]` if you pass an `X` with 784 features, and a `y` with 10 classes.

- **scales** – Scale of the randomly initialized weights. A list of floating point values. When you find good values for the scale of the weights you can speed up training a lot, and also improve performance. Defaults to *0.05*.
- **fan_outs** – Number of nonzero incoming connections to a hidden unit. Defaults to *None*, which means that all connections have non-zero weights.
- **output_act_funct** – Output activation function. Instance of type `Sigmoid`, `Linear`, `Softmax` from the `gdbn.activationFunctions` module. Defaults to `Softmax`.
- **real_valued_vis** – Set *True* (the default) if visible units are real-valued.
- **use_re_lu** – Set *True* to use rectified linear units. Defaults to *False*.
- **uniforms** – Not documented at this time.
- **learn_rates** – A list of learning rates, one entry per weight layer.

  An example: `[0.1, 0.1]`
- **learn_rate_decays** – The number with which the *learn_rate* is multiplied after each epoch of fine-tuning.
- **learn_rate_minimums** – The minimum *learn_rates*; after the learn rate reaches the minimum learn rate, the *learn_rate_decays* no longer has any effect.
- **momentum** – Momentum
- **l2_costs** – L2 costs per weight layer.
- **dropouts** – Dropouts per weight layer.
- **nesterov** – Not documented at this time.
- **nest_compare** – Not documented at this time.
- **rms_lims** – Not documented at this time.
- **learn_rates_pretrain** – A list of learning rates similar to *learn_rates_pretrain*, but used for pretraining. Defaults to value of *learn_rates* parameter.
- **momentum_pretrain** – Momentum for pre-training. Defaults to value of *momentum* parameter.
- **l2_costs_pretrain** – L2 costs per weight layer, for pre-training. Defaults to the value of *l2_costs* parameter.
- **nest_compare_pretrain** – Not documented at this time.
- **epochs** – Number of epochs to train (with backprop).
- **epochs_pretrain** – Number of epochs to pre-train (with CDN).
- **loss_funct** – A function that calculates the loss. Used for displaying learning progress and for `score()`.
- **minibatch_size** – Size of a minibatch.
- **minibatches_per_epoch** – Number of minibatches per epoch. The default is to use as many as fit into our training set.

- **pretrain_callback** – An optional function that takes as arguments the *DBN* instance, the epoch and the layer index as its argument, and is called for each epoch of pretraining.
- **fine_tune_callback** – An optional function that takes as arguments the *DBN* instance and the epoch, and is called for each epoch of fine tuning.
- **random_state** – An optional int used as the seed by the random number generator.
- **verbose** – Debugging output.

### 2.2.2 Example: MNIST

Let's train 2-layer neural network to do digit recognition on the MNIST dataset.

We first load the MNIST dataset, and split it up into a training and a test set:

```
from sklearn.cross_validation import train_test_split
from sklearn.datasets import fetch_mldata

mnist = fetch_mldata('MNIST original')
X_train, X_test, y_train, y_test = train_test_split(
    mnist.data / 255.0, mnist.target)
```

We then configure a neural network with 300 hidden units, a learning rate of `0.3` and a learning rate decay of `0.9`, which is the number that the learning rate will be multiplied with after each epoch.

```
from nolearn.dbn import DBN

clf = DBN(
    [X_train.shape[1], 300, 10],
    learn_rates=0.3,
    learn_rate_decays=0.9,
    epochs=10,
    verbose=1,
    )
```

Let us now train our network for 10 epochs. This will take around five minutes on a CPU:

```
clf.fit(X_train, y_train)
```

After training, we can use our trained neural network to predict the examples in the test set. We'll observe that our model has an accuracy of around **97.5%**.

```
from sklearn.metrics import classification_report
from sklearn.metrics import zero_one_score

y_pred = clf.predict(X_test)
print "Accuracy:", zero_one_score(y_test, y_pred)
print "Classification report:"
print classification_report(y_test, y_pred)
```

### 2.2.3 Example: Iris

In this example, we'll train a neural network for classification on the Iris flower data set. Due to the small number of examples, an SVM will typically perform better, but let us still see if our neural network is up to the task:

```
from sklearn.cross_validation import cross_val_score
from sklearn.datasets import load_iris
from sklearn.preprocessing import scale

iris = load_iris()
clf = DBN(
    [4, 4, 3],
    learn_rates=0.3,
    epochs=50,
    )

scores = cross_val_score(clf, scale(iris.data), iris.target, cv=10)
print "Accuracy: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() / 2)
```

This will print something like:

```
Accuracy: 0.97 (+/- 0.03)
```

### 2.2.4 Example: CIFAR-10

In this example, we'll train a neural network to do image classification using a subset of the CIFAR-10 dataset.

We assume that you have the Python version of the CIFAR-10 dataset downloaded and available in your working directory. We'll use only the first three batches of the dataset; the first two for training, the third one for testing.

Let us load the dataset:

```
import cPickle
import numpy as np

def load(name):
    with open(name, 'rb') as f:
        return cPickle.load(f)

dataset1 = load('data_batch_1')
dataset2 = load('data_batch_2')
dataset3 = load('data_batch_3')

data_train = np.vstack([dataset1['data'], dataset2['data']])
labels_train = np.hstack([dataset1['labels'], dataset2['labels']])

data_train = data_train.astype('float') / 255.
labels_train = labels_train
data_test = dataset3['data'].astype('float') / 255.
labels_test = np.array(dataset3['labels'])
```

We can now train our network. We'll configure the network so that it has 1024 units in the hidden layer, i.e. `[3072, 1024, 10]`. We'll train our network for 50 epochs, which will take a while if you're not using CUDAMat.

```
n_feat = data_train.shape[1]
n_targets = labels_train.max() + 1

net = DBN(
    [n_feat, n_feat / 3, n_targets],
    epochs=50,
    learn_rates=0.03,
    verbose=1,
```

```
    )
net.fit(data_train, labels_train)
```

Finally, we'll look at our network's performance:

```
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix

expected = labels_test
predicted = net.predict(data_test)

print "Classification report for classifier %s:\n%s\n" % (
    net, classification_report(expected, predicted))
print "Confusion matrix:\n%s" % confusion_matrix(expected, predicted)
```

You should see an f1-score of **0.49** and a confusion matrix that looks something like this:

```
  air aut bir cat dee dog fro hor shi tru
[[459  48  66  39  91  21   5  39 182  44]  airplane
 [ 28 584  12  31  23  22   8  29 117 188]  automobile
 [ 49  13 279 101 244 124  31  71  37  16]  bird
 [ 20  16  54 363 106 255  38  70  36  39]  cat
 [ 33  10  79  81 596  66  15  75  26   9]  deer
 [ 16  23  57 232 103 448  17  82  26  25]  dog
 [ 10  18  70 179 212 106 304  32  21  26]  frog
 [ 20   8  40  80 125  98  10 575  21  38]  horse
 [ 54  49  10  29  43  25   4   9 707  31]  ship
 [ 28 129   9  48  33  36  10  57 118 561]] truck
```

We should be able to improve on this score by using the full dataset and by training longer.

## 2.3 nolearn.decaf

### 2.3.1 API

**class** nolearn.decaf.**ConvNetFeatures**(*feature_layer='fc7_cudanet_out'*, *pretrained_params='imagenet.decafnet.epoch90'*, *pretrained_meta='imagenet.decafnet.meta'*, *center_only=True*, *classify_direct=False*, *verbose=0*)

Extract features from images using a pretrained ConvNet.

Based on Yangqing Jia and Jeff Donahue's DeCAF. Please make sure you read and accept DeCAF's license before you use this class.

If `classify_direct=False`, expects its input X to be a list of image filenames or arrays as produced by *np.array(Image.open(filename))*.

**__init__**(*feature_layer='fc7_cudanet_out'*, *pretrained_params='imagenet.decafnet.epoch90'*, *pretrained_meta='imagenet.decafnet.meta'*, *center_only=True*, *classify_direct=False*, *verbose=0*)

**Parameters**

- **feature_layer** – The ConvNet layer that's used for feature extraction. Defaults to *fc7_cudanet_out*. A description of all available layers for the ImageNet-1k-pretrained ConvNet is found in the DeCAF wiki. They are:
  - *pool5_cudanet_out*

- *fc6_cudanet_out*
- *fc6_neuron_cudanet_out*
- *fc7_cudanet_out*
- *fc7_neuron_cudanet_out*
- *probs_cudanet_out*

- **pretrained_params** – This must point to the file with the pretrained parameters. Defaults to *imagenet.decafnet.epoch90*. For the ImageNet-1k-pretrained ConvNet this file can be obtained from here: http://www.eecs.berkeley.edu/~jiayq/decaf_pretrained/
- **pretrained_meta** – Similar to *pretrained_params*, this must file to the file with the pretrained parameters' metadata. Defaults to *imagenet.decafnet.meta*.
- **center_only** – Use the center patch of the image only when extracting features. If *False*, use four corners, the image center and flipped variants and average a total of 10 feature vectors, which will usually yield better results. Defaults to *True*.
- **classify_direct** – When *True*, assume that input X is an array of shape (num x 256 x 256 x 3) as returned by *prepare_image*.

**prepare_image**(*image*)
Returns image of shape *(256, 256, 3)*, as expected by *transform* when *classify_direct = True*.

### 2.3.2 Installing DeCAF and downloading parameter files

You'll need to manually install DeCAF for *ConvNetFeatures* to work.

You will also need to download a tarball that contains pretrained parameter files from Yangqing Jia's homepage.

Refer to the location of the two files contained in the tarball when you instantiate *ConvNetFeatures* like so:

```
convnet = ConvNetFeatures(
    pretrained_params='/path/to/imagenet.decafnet.epoch90',
    pretrained_meta='/path/to/imagenet.decafnet.meta',
    )
```

For more information on how DeCAF works, please refer to [1].

### 2.3.3 Example: Dogs vs. Cats

What follows is a simple example that uses *ConvNetFeatures* and scikit-learn to classify images from the Kaggle Dogs vs. Cats challenge. Before you start, you must download the images from the Kaggle competition page. The `train/` folder will be referred to further down as `TRAIN_DATA_DIR`.

We'll first define a few imports and the paths to the files that we just downloaded:

```
import os

from nolearn.decaf import ConvNetFeatures
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
from sklearn.pipeline import Pipeline
from sklearn.utils import shuffle
```

[1] Jeff Donahue, Yangqing Jia, Oriol Vinyals, Judy Hoffman, Ning Zhang, Eric Tzeng, and Trevor Darrell. Decaf: A deep convolutional activation feature for generic visual recognition. arXiv preprint arXiv:1310.1531, 2013.

```
DECAF_IMAGENET_DIR = '/path/to/imagenet-files/'
TRAIN_DATA_DIR = '/path/to/dogs-vs-cats-training-images/'
```

A `get_dataset` function will return a list of all image filenames and labels, shuffled for our convenience:

```
def get_dataset():
    cat_dir = TRAIN_DATA_DIR + 'cat/'
    cat_filenames = [cat_dir + fn for fn in os.listdir(cat_dir)]
    dog_dir = TRAIN_DATA_DIR + 'dog/'
    dog_filenames = [dog_dir + fn for fn in os.listdir(dog_dir)]

    labels = [0] * len(cat_filenames) + [1] * len(dog_filenames)
    filenames = cat_filenames + dog_filenames
    return shuffle(filenames, labels, random_state=0)
```

We can now define our `sklearn.pipeline.Pipeline`, which merely consists of *`ConvNetFeatures`* and a `sklearn.linear_model.LogisticRegression` classifier.

```
def main():
    convnet = ConvNetFeatures(
        pretrained_params=DECAF_IMAGENET_DIR + 'imagenet.decafnet.epoch90',
        pretrained_meta=DECAF_IMAGENET_DIR + 'imagenet.decafnet.meta',
        )
    clf = LogisticRegression()
    pl = Pipeline([
        ('convnet', convnet),
        ('clf', clf),
        ])

    X, y = get_dataset()
    X_train, y_train = X[:100], y[:100]
    X_test, y_test = X[100:300], y[100:300]

    print "Fitting..."
    pl.fit(X_train, y_train)
    print "Predicting..."
    y_pred = pl.predict(X_test)
    print "Accuracy: %.3f" % accuracy_score(y_test, y_pred)

main()
```

Note that we use only 100 images to train our classifier (and 200 for testing). Regardless, and thanks to the magic of pre-trained convolutional nets, we're able to reach an accuracy of around 94%, which is an improvement of 11% over the classifier described in [2].

## 2.4 nolearn.inischema

`inischema` allows the definition of schemas for *.ini* configuration files.

Consider this sample schema:

```
>>> schema = '''
... [first]
... value1 = int
... value2 = string
```

[2] P. Golle. Machine learning attacks against the asirra captcha. In ACM CCS 2008, 2008.

```
... value3 = float
... value4 = listofstrings
... value5 = listofints
...
... [second]
... value1 = string
... value2 = int
... '''
```

This schema defines the sections, names and types of values expected in a schema file.

Using a concrete configuration, we can then use the schema to extract values:

```
>>> config = '''
... [first]
... value1 = 2
... value2 = three
... value3 = 4.4
... value4 = five six seven
... value5 = 8 9
...
... [second]
... value1 = ten
... value2 = 100
... value3 = what?
... '''

>>> result = parse_config(schema, config)
>>> from pprint import pprint
>>> pprint(result)
{'first': {'value1': 2,
           'value2': 'three',
           'value3': 4.4,
           'value4': ['five', 'six', 'seven'],
           'value5': [8, 9]},
 'second': {'value1': 'ten', 'value2': 100, 'value3': 'what?'}}
```

Values in the config file that are not in the schema are assumed to be strings.

This module is used in `nolearn.console` to allow for convenient passing of values from *.ini* files as function arguments for command line scripts.

`nolearn.inischema.`**`parse_config`**(*schema*, *config*)

## 2.5 nolearn.lasagne

Two introductory tutorials exist for *nolearn.lasagne*:

- Using convolutional neural nets to detect facial keypoints tutorial with code
- Training convolutional neural networks with nolearn

For specifics around classes and functions out of the *lasagne* package, such as layers, updates, and nonlinearities, you'll want to look at the Lasagne project's documentation.

*nolearn.lasagne* comes with a number of tests that demonstrate some of the more advanced features, such as networks with merge layers, and networks with multiple inputs.

Finally, there's a few presentations and examples from around the web. Note that some of these might need a specific version of nolearn and Lasange to run:

- Oliver Dürr's Convolutional Neural Nets II Hands On with code
- Roelof Pieters' presentation Python for Image Understanding comes with nolearn.lasagne code examples
- Benjamin Bossan's Otto Group Product Classification Challenge using nolearn/lasagne
- Kaggle's instructions on how to set up an AWS GPU instance to run nolearn.lasagne and the facial keypoint detection tutorial
- An example convolutional autoencoder
- Winners of the saliency prediction task in the 2015 LSUN Challenge have published their lasagne/nolearn-based code.

### 2.5.1 API

**class** nolearn.lasagne.**NeuralNet**(*layers*, *update=<function nesterov_momentum>*, *loss=None*, *objective=<function objective>*, *objective_loss_function=None*, *batch_iterator_train=<nolearn.lasagne.base.BatchIterator object>*, *batch_iterator_test=<nolearn.lasagne.base.BatchIterator object>*, *regression=False*, *max_epochs=100*, *train_split=<nolearn.lasagne.base.TrainSplit object>*, *custom_scores=None*, *scores_train=None*, *scores_valid=None*, *X_tensor_type=None*, *y_tensor_type=None*, *use_label_encoder=False*, *on_batch_finished=None*, *on_epoch_finished=None*, *on_training_started=None*, *on_training_finished=None*, *more_params=None*, *check_input=True*, *verbose=0*, ***kwargs*)

A scikit-learn estimator based on Lasagne.

**class** nolearn.lasagne.**BatchIterator**(*batch_size*, *shuffle=False*, *seed=42*)

**class** nolearn.lasagne.**TrainSplit**(*eval_size*, *stratify=True*)

## 2.6 nolearn.metrics

nolearn.metrics.**multiclass_logloss**(*actual*, *predicted*, *eps=1e-15*)

Multi class version of Logarithmic Loss metric.

**Parameters**

- **actual** – Array containing the actual target classes
- **predicted** – Matrix with class predictions, one probability per class

nolearn.metrics.**learning_curve**(*self*, *dataset*, *classifier*, *steps=10*, *verbose=0*, *random_state=42*)

Create a learning curve that uses more training cases with each step.

**Parameters**

- **dataset** (Dataset) – Dataset to work with
- **classifier** (BaseEstimator) – Classifier for fitting and making predictions.
- **steps** (*int*) – Number of steps in the learning curve.

**Result** 3-tuple with lists *scores_train*, *scores_test*, *sizes*

Drawing the resulting learning curve can be done like this:

```
dataset = Dataset()
clf = LogisticRegression()
scores_train, scores_test, sizes = learning_curve(dataset, clf)
pl.plot(sizes, scores_train, 'b', label='training set')
pl.plot(sizes, scores_test, 'r', label='test set')
pl.legend(loc='lower right')
pl.show()
```

nolearn.metrics.**learning_curve_logloss**(*self*, *dataset*, *classifier*, *steps=10*, *verbose=0*, *random_state=42*)

Same as *learning_curve()* but uses *multiclass_logloss()* as the loss funtion.

CHAPTER 3

# Indices and tables

- genindex
- modindex
- search

## n

## Symbols



## B



## C



## D



## L



## M



## N



## P



## T