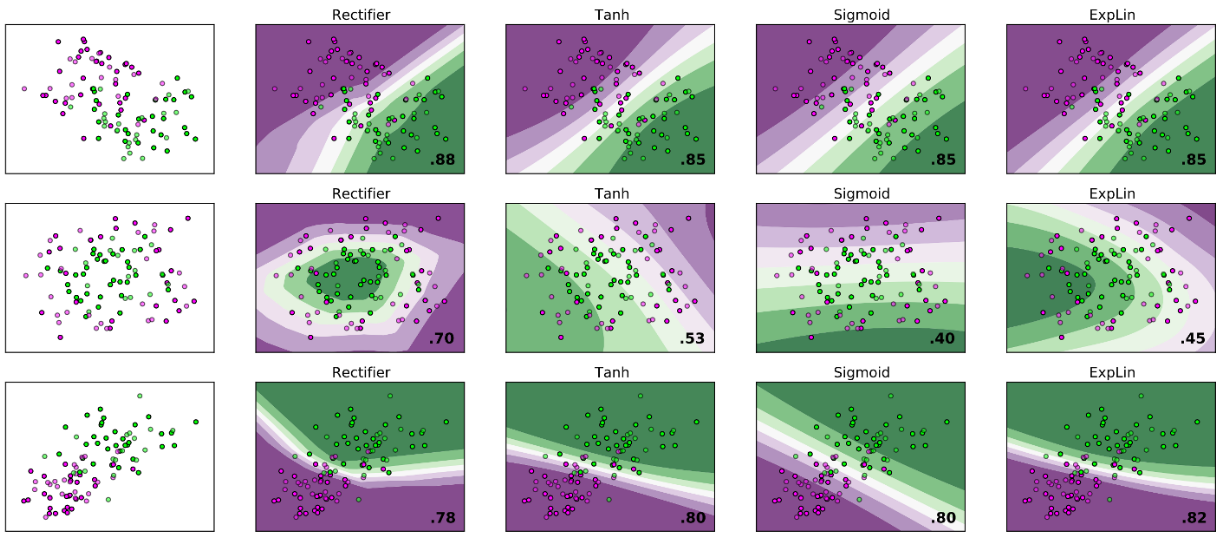
scikit-neuralnetwork

Release 0.7

Sep 04, 2017

Contents

Deep neural network implementation without the learning cliff! This library implements multi-layer perceptrons as a wrapper for the powerful `pylearn2` library that's compatible with `scikit-learn` for a more user-friendly and Pythonic interface.



CHAPTER 1

Module Reference

`sknn.mlp` — Multi-Layer Perceptrons

In this module, a neural network is made up of multiple layers — hence the name multi-layer perceptron! You need to specify these layers by instantiating one of two types of specifications:

- *`sknn.mlp.Layer`*: A standard feed-forward layer that can use linear or non-linear activations.
- *`sknn.mlp.Convolution`*: An image-based convolve operation with shared weights, linear or not.

In practice, you need to create a list of these specifications and provide them as the `layers` parameter to the *`sknn.mlp.Regressor`* or *`sknn.mlp.Classifier`* constructors.

Layer Specifications

class `sknn.mlp.`**`Layer`**(*type*, *warning=None*, *name=None*, *units=None*, *weight_decay=None*, *dropout=None*, *normalize=None*, *frozen=False*)

Specification for a layer to be passed to the neural network during construction. This includes a variety of parameters to configure each layer based on its activation type.

Parameters type: str

Select which activation function this layer should use, as a string. Specifically, options are `Rectifier`, `Sigmoid`, `Tanh`, and `ExpLin` for non-linear layers and `Linear` or `Softmax` for output layers.

name: str, optional

You optionally can specify a name for this layer, and its parameters will then be accessible to scikit-learn via a nested sub-object. For example, if name is set to `layer1`, then the parameter `layer1__units` from the network is bound to this layer's `units` variable.

The name defaults to `hiddenN` where N is the integer index of that layer, and the final layer is always `output` without an index.

units: int

The number of units (also known as neurons) in this layer. This applies to all layer types except for convolution.

weight_decay: float, optional

The coefficient for L1 or L2 regularization of the weights. For example, a value of 0.0001 is multiplied by the L1 or L2 weight decay equation.

dropout: float, optional

The ratio of inputs to drop out for this layer during training. For example, 0.25 means that 25% of the inputs will be excluded for each training sample, with the remaining inputs being renormalized accordingly.

normalize: str, optional

Enable normalization of this layer. Can be either *batch* for batch normalization or (soon) *weights* for weight normalization. Default is no normalization.

frozen: bool, optional

Specify whether to freeze a layer's parameters so they are not adjusted during the training. This is useful when relying on pre-trained neural networks.

warning: None

You should use keyword arguments after *type* when initializing this object. If not, the code will raise an AssertionError.

Methods

`set_params`

class `sknn.mlp.`**`Convolution`**(*type*, *warning=None*, *name=None*, *channels=None*, *kernel_shape=None*, *kernel_stride=None*, *border_mode=u'valid'*, *pool_shape=None*, *pool_type=None*, *scale_factor=None*, *weight_decay=None*, *dropout=None*, *normalize=None*, *frozen=False*)

Specification for a convolution layer to be passed to the neural network in construction. This includes a variety of convolution-specific parameters to configure each layer, as well as activation-specific parameters.

Parameters type: str

Select which activation function this convolution layer should use, as a string. For hidden layers, you can use the following convolution types `Rectifier`, `ExpLin`, `Sigmoid`, `Tanh` or `Linear`.

name: str, optional

You optionally can specify a name for this layer, and its parameters will then be accessible to scikit-learn via a nested sub-object. For example, if name is set to `layer1`, then the parameter `layer1__units` from the network is bound to this layer's `units` variable.

The name defaults to `hiddenN` where N is the integer index of that layer, and the final layer is always `output` without an index.

channels: int

Number of output channels for the convolution layers. Each channel has its own set of shared weights which are trained by applying the kernel over the image.

kernel_shape: tuple of ints

A two-dimensional tuple of integers corresponding to the shape of the kernel when convolution is used. For example, this could be a square kernel *(3,3)* or a full horizontal or vertical kernel on the input matrix, e.g. *(N,1)* or *(1,N)*.

kernel_stride: tuple of ints, optional

A two-dimensional tuple of integers that represents the steps taken by the kernel through the input image. By default, this is set to *(1,1)* and can be customized separately to pooling.

border_mode: str

String indicating the way borders in the image should be processed, one of two options:

- *valid* — Only pixels from input where the kernel fits within bounds are processed.
- *full* — All pixels from input are processed, and the boundaries are zero-padded.
- *same* — The output resolution is set to the exact same as the input.

The size of the output will depend on this mode, for *full* it's identical to the input, but for *valid* (default) it will be smaller or equal.

pool_shape: tuple of ints, optional

A two-dimensional tuple of integers corresponding to the pool size for downsampling. This should be square, for example *(2,2)* to reduce the size by half, or *(4,4)* to make the output a quarter of the original.

Pooling is applied after the convolution and calculation of its activation.

pool_type: str, optional

Type of the pooling to be used; can be either *max* or *mean*. If a *pool_shape* is specified the default is to take the maximum value of all inputs that fall into this pool. Otherwise, the default is None and no pooling is used for performance.

scale_factor: tuple of ints, optional

A two-dimensional tuple of integers corresponding to upscaling ration. This should be square, for example *(2,2)* to increase the size by double, or *(4,4)* to make the output four times the original.

Upscaling is applied before the convolution and calculation of its activation.

weight_decay: float, optional

The coefficient for L1 or L2 regularization of the weights. For example, a value of 0.0001 is multiplied by the L1 or L2 weight decay equation.

dropout: float, optional

The ratio of inputs to drop out for this layer during training. For example, 0.25 means that 25% of the inputs will be excluded for each training sample, with the remaining inputs being renormalized accordingly.

normalize: str, optional

Enable normalization of this layer. Can be either *batch* for batch normalization or (soon) *weights* for weight normalization. Default is no normalization.

frozen: bool, optional

Specify whether to freeze a layer's parameters so they are not adjusted during the training. This is useful when relying on pre-trained neural networks.

warning: None

You should use keyword arguments after *type* when initializing this object. If not, the code will raise an AssertionError.

Methods

`set_params`

MultiLayerPerceptron

Most of the functionality provided to simulate and train multi-layer perceptron is implemented in the (abstract) class *`sknn.mlp.MultiLayerPerceptron`*. This class documents all the construction parameters for Regressor and Classifier derived classes (see below), as well as their various helper functions.

class `sknn.mlp.`**`MultiLayerPerceptron`**(*layers*, *warning=None*, *parameters=None*, *random_state=None*, *learning_rule=u'sgd'*, *learning_rate=0.01*, *learning_momentum=0.9*, *normalize=None*, *regularize=None*, *weight_decay=None*, *dropout_rate=None*, *batch_size=1*, *n_iter=None*, *n_stable=10*, *f_stable=0.001*, *valid_set=None*, *valid_size=0.0*, *loss_type=None*, *callback=None*, *debug=False*, *verbose=None*, ***params*)

Abstract base class for wrapping all neural network functionality from PyLearn2, common to multi-layer perceptrons in `sknn.mlp` and auto-encoders in in `sknn.ae`.

Parameters layers: list of Layer

An iterable sequence of each layer each as a *`sknn.mlp.Layer`* instance that contains its type, optional name, and any paramaters required.

- For hidden layers, you can use the following layer types: `Rectifier`, `ExpLin`, `Sigmoid`, `Tanh`, or `Convolution`.
- For output layers, you can use the following layer types: `Linear` or `Softmax`.

It's possible to mix and match any of the layer types, though most often you should probably use hidden and output types as recommended here. Typically, the last entry in this `layers` list should contain `Linear` for regression, or `Softmax` for classification.

random_state: int, optional

Seed for the initialization of the neural network parameters (e.g. weights and biases). This is fully deterministic.

parameters: list of tuple of array-like, optional

A list of `(weights, biases)` tuples to be reloaded for each layer, in the same order as `layers` was specified. Useful for initializing with pre-trained networks.

learning_rule: str, optional

Name of the learning rule used during stochastic gradient descent, one of `sgd`, `momentum`, `nesterov`, `adadelta`, `adagrad` or `rmsprop` at the moment. The default is vanilla `sgd`.

learning_rate: float, optional

Real number indicating the default/starting rate of adjustment for the weights during gradient descent. Different learning rules may take this into account differently. Default is `0.01`.

learning_momentum: float, optional

Real number indicating the momentum factor to be used for the learning rule 'momentum'. Default is `0.9`.

batch_size: int, optional

Number of training samples to group together when performing stochastic gradient descent (technically, a "minibatch"). By default each sample is treated on its own, with `batch_size=1`. Larger batches are usually faster.

n_iter: int, optional

The number of iterations of gradient descent to perform on the neural network's weights when training with `fit()`.

n_stable: int, optional

Number of interations after which training should return when the validation error remains (near) constant. This is usually a sign that the data has been fitted, or that optimization may have stalled. If no validation set is specified, then stability is judged based on the training error. Default is `10`.

f_stable: float, optional

Threshold under which the validation error change is assumed to be stable, to be used in combination with *n_stable*. This is calculated as a relative ratio of improvement, so if the results are only 0.1% better training is considered stable. The training set is used as fallback if there's no validation set. Default is ''0.001'.

valid_set: tuple of array-like, optional

Validation set (X_v, y_v) to be used explicitly while training. Both arrays should have the same size for the first dimention, and the second dimention should match with the training data specified in `fit()`.

valid_size: float, optional

Ratio of the training data to be used for validation. 0.0 means no validation, and 1.0 would mean there's no training data! Common values are 0.1 or 0.25.

normalize: string, optional

Enable normalization for all layers. Can be either *batch* for batch normalization or (soon) *weights* for weight normalization. Default is no normalization.

regularize: string, optional

Which regularization technique to use on the weights, for example `L2` (most common) or `L1` (quite rare), as well as `dropout`. By default, there's no regularization, unless another parameter implies it should be enabled, e.g. if `weight_decay` or `dropout_rate` are specified.

weight_decay: float, optional

The coefficient used to multiply either `L1` or `L2` equations when computing the weight decay for regularization. If `regularize` is specified, this defaults to 0.0001.

dropout_rate: float, optional

What rate to use for drop-out training in the inputs (jittering) and the hidden layers, for each training example. Specify this as a ratio of inputs to be randomly excluded during training, e.g. 0.75 means only 25% of inputs will be included in the training.

loss_type: string, optional

The cost function to use when training the network. There are two valid options:

- `mse` — Use mean squared error, for learning to predict the mean of the data.
- `mae` — Use mean average error, for learning to predict the median of the data.
- `mcc` — Use mean categorical cross-entropy, particularly for classifiers.

The default option is `mse` for regressors and `mcc` for classifiers, but `mae` can only be applied to layers of type `Linear` or `Gaussian` and they must be used as the output layer (PyLearn2 only).

callback: callable or dict, optional

An observer mechanism that exposes information about the inner training loop. This is either a single function that takes `cbs(event, **variables)` as a parameter, or a dictionary of functions indexed by on *event* string that conforms to `cb(**variables)`.

There are multiple events sent from the inner training loop:

- `on_train_start` — Called when the main training function is entered.
- `on_epoch_start` — Called the first thing when a new iteration starts.
- `on_batch_start` — Called before an individual batch is processed.
- `on_batch_finish` — Called after that individual batch is processed.
- `on_epoch_finish` — Called the first last when the iteration is done.
- `on_train_finish` — Called just before the training function exits.

For each function, the `variables` dictionary passed contains all local variables within the training implementation.

debug: bool, optional

Should the underlying training algorithms perform validation on the data as it's optimizing the model? This makes things slower, but errors can be caught more effectively. Default is off.

verbose: bool, optional

How to initialize the logging to display the results during training. If there is already a logger initialized, either `sknn` or the root logger, then this function does nothing. Otherwise:

- `False` — Setup new logger that shows only warnings and errors.
- `True` — Setup a new logger that displays all debug messages.
- `None` — Don't setup a new logger under any condition (default).

Using the built-in python `logging` module, you can control the detail and style of output by customising the verbosity level and formatter for `sknn` logger.

warning: None

You should use keyword arguments after *layers* when initializing this object. If not, the code will raise an AssertionError.

Attributes

`is_classifier`
`is_initialized`

Methods

`get_parameters`
`get_params`
`is_convolution`
`set_parameters`

When using the multi-layer perceptron, you should initialize a Regressor or a Classifier directly.

Regressor

See the class *`sknn.mlp.MultiLayerPerceptron`* for inherited construction parameters.

class `sknn.mlp.`**`Regressor`**(*layers*, *warning=None*, *parameters=None*, *random_state=None*, *learning_rule=u'sgd'*, *learning_rate=0.01*, *learning_momentum=0.9*, *normalize=None*, *regularize=None*, *weight_decay=None*, *dropout_rate=None*, *batch_size=1*, *n_iter=None*, *n_stable=10*, *f_stable=0.001*, *valid_set=None*, *valid_size=0.0*, *loss_type=None*, *callback=None*, *debug=False*, *verbose=None*, ***params*)

Attributes

`is_classifier`
`is_initialized`

Methods

`fit`
`get_parameters`
`get_params`
`is_convolution`
`predict`
`set_parameters`

`fit`(*X*, *y*, *w=None*)

Fit the neural network to the given continuous data as a regression problem.

Parameters **X** : array-like, shape (n_samples, n_inputs)

Training vectors as real numbers, where n_samples is the number of samples and n_inputs is the number of input features.

y : array-like, shape (n_samples, n_outputs)

Target values are real numbers used as regression targets.

w : array-like (optional), shape (n_samples)

Floating point weights for each of the training samples, used as mask to modify the cost function during optimization.

Returns **self** : object

Returns this instance.

`get_parameters``()`

Extract the neural networks weights and biases layer by layer. Only valid once the neural network has been initialized, for example via *fit()* function.

Returns **params** : list of tuples

For each layer in the order they are passed to the constructor, a named-tuple of three items *weights*, *biases* (both numpy arrays) and *name* (string) in that order.

`is_convolution`(*input=None*, *output=False*)

Check whether this neural network includes convolution layers in the first or last position.

Parameters **input** : boolean, optional

Whether the first layer should be checked for convolution. Default True.

output : boolean, optional

Whether the last layer should be checked for convolution. Default False.

Returns **is_conv** : boolean

True if either of the specified layers are indeed convolution, False otherwise.

`is_initialized`

Check if the neural network was setup already.

`predict`(*X*)

Calculate predictions for specified inputs.

Parameters **X** : array-like, shape (n_samples, n_inputs)

The input samples as real numbers.

Returns **y** : array, shape (n_samples, n_outputs)

The predicted values as real numbers.

`set_parameters`(*storage*)

Store the given weighs and biases into the neural network. If the neural network has not been initialized, use the *weights* list as construction parameter instead. Otherwise if the neural network is initialized, this function will extract the parameters from the input list or dictionary and store them accordingly.

Parameters **storage** : list of tuples, or dictionary of tuples

Either a list of tuples for each layer, storing two items *weights* and *biases* in the exact same order as construction. Alternatively, if this is a dictionary, a string to tuple mapping for each layer also storing *weights* and *biases* but not necessarily for all layers.

Classifier

Also check the `sknn.mlp.MultiLayerPerceptron` class for inherited construction parameters.

class `sknn.mlp.`**`Classifier`**(*layers*, *warning=None*, *parameters=None*, *random_state=None*, *learning_rule=u'sgd'*, *learning_rate=0.01*, *learning_momentum=0.9*, *normalize=None*, *regularize=None*, *weight_decay=None*, *dropout_rate=None*, *batch_size=1*, *n_iter=None*, *n_stable=10*, *f_stable=0.001*, *valid_set=None*, *valid_size=0.0*, *loss_type=None*, *callback=None*, *debug=False*, *verbose=None*, ***params*)

Attributes

`classes_`
`is_classifier`
`is_initialized`

Methods

`fit`
`get_parameters`
`get_params`
`is_convolution`
`partial_fit`
`predict`
`predict_proba`
`set_parameters`

`classes_`

Return a list of class labels used for each feature. For single feature classification, the index of the label in the array is the same as returned by *predict_proba()* (e.g. labels *[-1, 0, +1]* mean indices *[0, 1, 2]*).

In the case of multiple feature classification, the index of the label must be offset by the number of labels for previous features. For example, if the second feature also has labels *[-1, 0, +1]* its indicies will be *[3, 4, 5]* resuming from the first feature in the array returned by *predict_proba()*.

Returns **c** : list of array, shape (n_classes, n_labels)

List of the labels as integers used for each feature.

`fit`(*X*, *y*, *w=None*)

Fit the neural network to symbolic labels as a classification problem.

Parameters **X** : array-like, shape (n_samples, n_features)

Training vectors as real numbers, where n_samples is the number of samples and n_inputs is the number of input features.

y : array-like, shape (n_samples, n_classes)

Target values as integer symbols, for either single- or multi-output classification problems.

w : array-like (optional), shape (n_samples)

Floating point weights for each of the training samples, used as mask to modify the cost function during optimization.

Returns self : object

Returns this instance.

get_parameters()

Extract the neural networks weights and biases layer by layer. Only valid once the neural network has been initialized, for example via *fit()* function.

Returns params : list of tuples

For each layer in the order they are passed to the constructor, a named-tuple of three items *weights*, *biases* (both numpy arrays) and *name* (string) in that order.

is_convolution(*input=None*, *output=False*)

Check whether this neural network includes convolution layers in the first or last position.

Parameters input : boolean, optional

Whether the first layer should be checked for convolution. Default True.

output : boolean, optional

Whether the last layer should be checked for convolution. Default False.

Returns is_conv : boolean

True if either of the specified layers are indeed convolution, False otherwise.

is_initialized

Check if the neural network was setup already.

predict(*X*)

Predict class by converting the problem to a regression problem.

Parameters X : array-like of shape (n_samples, n_features)

The input data.

Returns y : array-like, shape (n_samples,) or (n_samples, n_classes)

The predicted classes, or the predicted values.

predict_proba(*X*, *collapse=True*)

Calculate probability estimates based on these input features.

Parameters X : array-like of shape [n_samples, n_features]

The input data as a numpy array.

Returns y_prob : list of arrays of shape [n_samples, n_features, n_classes]

The predicted probability of the sample for each class in the model, in the same order as the classes.

set_parameters(*storage*)

Store the given weighs and biases into the neural network. If the neural network has not been initialized, use the *weights* list as construction parameter instead. Otherwise if the neural network is initialized, this function will extract the parameters from the input list or dictionary and store them accordingly.

Parameters storage : list of tuples, or dictionary of tuples

Either a list of tuples for each layer, storing two items *weights* and *biases* in the exact same order as construction. Alternatively, if this is a dictionary, a string to tuple mapping for each layer also storing *weights* and *biases* but not necessarily for all layers.

CHAPTER 2

User Guide

Installation

You have multiple options to get up and running, though using `pip` is by far the easiest and most reliable.

A) Download Latest Release [Recommended]

If you want to use the latest official release, you can do so from PYPI directly:

```
> pip install scikit-neuralnetwork
```

This will install the latest official `Lasagne` and `Theano` as well as other minor packages too as a dependency. We strongly suggest you use a virtualenv for Python.

B) Pulling Repositories [Optional]

If you want to use the more advanced features like convolution, pooling or upscaling, these depend on the latest code from `Lasagne` and `Theano` master branches. You can install them manually as follows:

```
> pip install -r https://raw.githubusercontent.com/aigamedev/scikit-neuralnetwork/
↪master/requirements.txt
```

Once that's done, you can grab this repository and install from `setup.py` in the exact same way:

```
> git clone https://github.com/aigamedev/scikit-neuralnetwork.git
> cd scikit-neuralnetwork; python setup.py develop
```

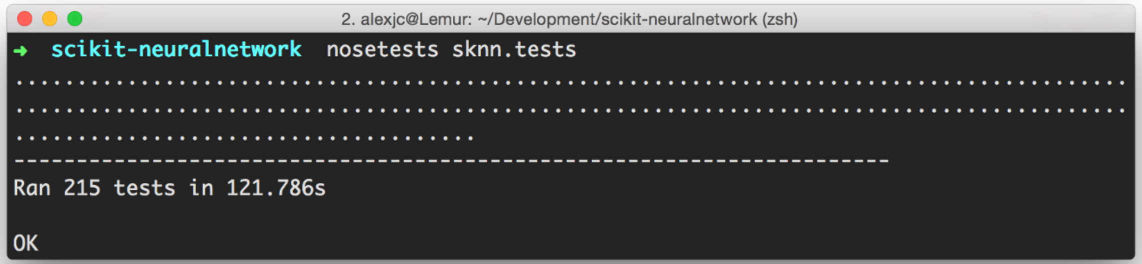
This will make the `sknn` package globally available within Python as a reference to the current directory.

Running Tests

We encourage you to launch the tests to check everything is working using the following commands:

```
> pip install nose
> nosetests -v sknn
```

Use the additional command-line parameters in the test runner `--processes=8` and `--process-timeout=60` to speed things up on powerful machines. The result should look as follows in your terminal.



```
➜  scikit-neuralnetwork  nosetests sknn.tests
...........................................................................................
...........................................................................................
.....................................
----------------------------------------------------------------------
Ran 215 tests in 121.786s

OK
```

We strive to maintain 100% test coverage for all code-paths, to ensure that rapid changes in the underlying `Lasagne` and `Theano` libraries are caught automatically.

Model Setup & Training

Regression

Assuming your data is in the form of `numpy.ndarray` stored in the variables `X_train` and `y_train` you can train a *`sknn.mlp.Regressor`* neural network. The input and output arrays are continuous values in this case, but it's best if you normalize or standardize your inputs to the `[0..1]` or `[-1..1]` range. (See the *sklearn Pipeline* example below.)

```
from sknn.mlp import Regressor, Layer

nn = Regressor(
    layers=[
        Layer("Rectifier", units=100),
        Layer("Linear")],
    learning_rate=0.02,
    n_iter=10)
nn.fit(X_train, y_train)
```

This will train the regressor for 10 epochs (specified via the `n_iter` parameter). The `layers` parameter specifies how the neural network is structured; see the *`sknn.mlp.Layer`* documentation for supported layer types and parameters.

Then you can use the trained NN as follows:

```
y_example = nn.predict(X_example)
```

This will return a new `numpy.ndarray` with the results of the feed-forward simulation of the network and the estimates given the input features.

Classification

If your data in `numpy.ndarray` contains integer labels as outputs and you want to train a neural network to classify the data, use the following snippet:

```
from sknn.mlp import Classifier, Layer

nn = Classifier(
    layers=[
        Layer("Maxout", units=100, pieces=2),
        Layer("Softmax")],
    learning_rate=0.001,
    n_iter=25)
nn.fit(X_train, y_train)
```

It's also a good idea to normalize or standardize your data in this case too, for example using a *sklearn Pipeline* below. The code here will train for 25 iterations. Note that a `Softmax` output layer activation type is used here, and it's recommended as a default for classification problems.

If you want to do multi-label classification, simply fit using a `y` array of integers that has multiple dimensions, e.g. shape *(N, 3)* for three different classes. Then, make sure the last layer is `Sigmoid` instead.

```
y_example = nn.predict(X_example)
```

This code will run the classification with the neural network, and return a list of labels predicted for each of the example inputs. If you need to access the probabilities for the predictions, use `predict_proba()` and see the content of the `classes_` property that provides the labels for each features, which you can use to compute the probability indices.

Convolution

Working with images as inputs in 2D (as greyscale) or 3D (as RGB) images stored in `numpy.ndarray`, you can use convolution to train a neural network with shared weights. Here's an example how classification would work:

```
from sknn.mlp import Classifier, Convolution, Layer

nn = Classifier(
    layers=[
        Convolution("Rectifier", channels=8, kernel_shape=(3,3)),
        Layer("Softmax")],
    learning_rate=0.02,
    n_iter=5)
nn.fit(X_train, y_train)
```

The neural network here is trained with eight kernels of shared weights in a `3x3` matrix, each outputting to its own channel. The rest of the code remains the same, but see the *`sknn.mlp.Layer`* documentation for supported convolution layer types and parameters.

Per-Sample Weighting

When training a classifier with data that has unbalanced labels, it's useful to adjust the weight of the different training samples to prevent bias. This is achieved via a feature called masking. You can specify the weights of each training sample when calling the `fit()` function.

```
w_train = numpy.array((X_train.shape[0],))
w_train[y_train == 0] = 1.2
```

```
w_train[y_train == 1] = 0.8

nn.fit(X_train, y_train, w_train)
```

In this case, there are two classes `0` given weight `1.2`, and `1` with weighting `0.8`. This feature also works for regressors as well.

Native & Custom Layers

In case you want to use more advanced features not directly supported by `scikit-neuralnetwork`, you can use so-called `sknn.nn.Native` layers that are handled directly by the backend. This allows you to use all features from the Lasagne library, for example.

```
from lasagne import layers as lasagne, nonlinearities as nl
from sknn.mlp import Classifier, Layer, Native

nn = Classifier(layers=[
        Native(lasagne.DenseLayer, num_units=256, nonlinearity=nl.leaky_rectify),
        Layer("Linear")])
```

When you insert a `Native` specification into the `layers` list, the first parameter is a constructor or class type that builds an object to insert into the network. In the example above, it's a `lasagne.layers.DenseLayer`. The keyword parameters (e.g. `nonlinearity`) are passed to this constructor dynamically when the network is initialized.

You can use this feature to implement recurrent layers like LSTM or GRU, and any other features not directly supported. Keep in mind that this may affect compatibility in future releases, and also may expose edge cases in the code (e.g. serialization, determinism).

Low-Level Configuration

Keyboard Interrupt

If you want to manually interrupt the main training loop by pressing `CTRL+C` but still finish the rest of your training script, you can wrap the call to fit with an exception handler:

```
# Setup experiment model and data.
nn = mlp.Regressor(...)

# Perform the gradient descent training.
try:
    nn.fit(X, y)
except KeyboardInterrupt:
    pass

# Finalize the experiment here.
print('score =', nn.score(X, y))
```

This was designed to work with both multi-layer perceptrons in `sknn.mlp` and auto-encoders in `sknn.ae`.

CPU vs. GPU Platform

To setup the library to use your GPU or CPU explicitly in 32-bit or 64-bit mode, you can use the `platform` pseudo-module. It's a syntactic helper to setup the `THEANO_FLAGS` environment variable in a Pythonic way, for example:

```
# Use the GPU in 32-bit mode, falling back otherwise.
from sknn.platform import gpu32

# Use the CPU in 64-bit mode.
from sknn.platform import cpu64
```

WARNING: This will only work if your program has not yet imported the `theano` module, due to the way that library is designed. If `THEANO_FLAGS` are set on the command-line, they are not overridden.

Multiple Threads

In CPU mode and on supported platforms (e.g. gcc on Linux), to use multiple threads (by default the number of processors) you can also import from the `platform` pseudo-module as follows:

```
# Use the maximum number of threads for this script.
from sknn.platform import cpu32, threading
```

If you want to specify the number of threads exactly, you can import for example `threads2` or `threads8` — or any other positive number that's supported by your OS. Alternatively, you can manually set these values by using the `OMP_NUM_THREADS` environment variable directly, and setting `THEANO_FLAGS` to include `openmp=True`.

Backend Configuration

As of version 0.3, `scikit-neuralnetwork` supports multiple neural network implementations called backends, each wrapped behind an identical standardized interface. To configure a backend, you can do so by importing the corresponding module:

```
from sknn.backend import lasagne
```

As long as you call this before creating a neural network, this will register the PyLearn2 implementation as the one that's used. Supported backends are currently `lasagne` (default) and `pylearn2` (removed).

Input / Output

Verbose Mode

To see the output of the neural network's training, configure the Python logger called `sknn` or the default root logger. This is possible using the standard `logging` module which you can setup as follows:

```
import sys
import logging

logging.basicConfig(
            format="%(message)s",
            level=logging.DEBUG,
            stream=sys.stdout)
```

Change the log level to `logging.INFO` for less information about each epoch, or `logging.WARNING` only to receive messages about problems or failures.

Using the flag `verbose=True` on either *`sknn.mlp.Classifier`* and *`sknn.mlp.Regressor`* will setup a default logger at `DEBUG` level if it does not exist, and `verbose=False` will setup a default logger at level `WARNING` if no logging has been configured.

Saving & Loading

To save a trained neural network to disk, you can do the following after having initialized your multi-layer perceptron as the variable `nn` and trained it:

```
import pickle
pickle.dump(nn, open('nn.pkl', 'wb'))
```

After this, the file `nn.pkl` will be available in the current working directory — which you can reload at any time:

```
import pickle
nn = pickle.load(open('nn.pkl', 'rb'))
```

In this case, you can use the reloaded multi-layer perceptron as if it had just been trained. This will also work on different machines, whether CPU or GPU.

NOTE: You can serialize complex pipelines (for example from this section *sklearn Pipeline*) using this exact same approach.

Extracting Parameters

To access the weights and biases from the neural network layers, you can call the following function on any initialized neural network:

```
> nn.get_parameters()
[Parameters(layer='hidden0', weights=array([[...]]), biases=array([[...]])),
 Parameters(layer='output', weights=array(...), biases=array(...))]
```

The list is ordered in the same way as the `layers` parameter passed to the constructor. Each item in the list is a named-tuple with `names` (string), `weights` and `biases` (both numpy.array).

scikit-learn Features

The examples in this section help you get more out of `scikit-neuralnetwork`, in particular via its integration with `scikit-learn`.

sklearn Pipeline

Typically, neural networks perform better when their inputs have been normalized or standardized. Using a scikit-learn's pipeline support is an obvious choice to do this.

Here's how to setup such a pipeline with a multi-layer perceptron as a classifier:

```
from sknn.mlp import Classifier, Layer

from sklearn.pipeline import Pipeline
from sklearn.preprocessing import MinMaxScaler

pipeline = Pipeline([
        ('min/max scaler', MinMaxScaler(feature_range=(0.0, 1.0))),
        ('neural network', Classifier(layers=[Layer("Softmax")], n_iter=25))])
pipeline.fit(X_train, y_train)
```

You can then use the pipeline as you would the neural network, or any other standard API from scikit-learn.

Grid Search

In scikit-learn, you can use a `GridSearchCV` to optimize your neural network's hyper-parameters automatically, both the top-level parameters and the parameters within the layers. For example, assuming you have your MLP constructed as in the *Regression* example in the local variable called `nn`, the layers are named automatically so you can refer to them as follows:

- `hidden0`
- `hidden1`
- ...
- `output`

Keep in mind you can manually specify the `name` of any `Layer` in the constructor if you don't want the automatically assigned name. Then, you can use sklearn's hierarchical parameters to perform a grid search over those nested parameters too:

```
from sklearn.grid_search import GridSearchCV

gs = GridSearchCV(nn, param_grid={
    'learning_rate': [0.05, 0.01, 0.005, 0.001],
    'hidden0__units': [4, 8, 12],
    'hidden0__type': ["Rectifier", "Sigmoid", "Tanh"]})
gs.fit(X, y)
```

This will search through the listed `learning_rate` values, the number of hidden units and the activation type for that layer too, and find the best combination of parameters.

Randomized Search

In the cases when you have large numbers of hyper-parameters that you want to try automatically to find a good combination, you can use a randomized search as follows:

```
from scipy import stats
from sklearn.grid_search import RandomizedSearchCV

rs = RandomizedSearchCV(nn, param_distributions={
    'learning_rate': stats.uniform(0.001, 0.05),
    'hidden0__units': stats.randint(4, 12),
    'hidden0__type': ["Rectifier", "Sigmoid", "Tanh"]})
rs.fit(X, y)
```

This works for both *`sknn.mlp.Classifier`* and *`sknn.mlp.Regressor`*.

Unsupervised Pre-Training

(**NOTE**: This is currently not supported with the Lasagne backend.)

If you have large quantities of unlabeled data, you may benefit from pre-training using an auto-encoder style architecture in an unsupervised learning fashion.

```
from sknn import ae, mlp

# Initialize auto-encoder for unsupervised learning.
myae = ae.AutoEncoder(
            layers=[
                ae.Layer("Tanh", units=128),
                ae.Layer("Sigmoid", units=64)],
            learning_rate=0.002,
            n_iter=10)

# Layerwise pre-training using only the input data.
myae.fit(X)

# Initialize the multi-layer perceptron with same base layers.
mymlp = mlp.Regressor(
            layers=[
                mlp.Layer("Tanh", units=128),
                mlp.Layer("Sigmoid", units=64),
                mlp.Layer("Linear")])

# Transfer the weights from the auto-encoder.
myae.transfer(mymlp)
# Now perform supervised-learning as usual.
mymlp.fit(X, y)
```

The downside of this approach is that auto-encoders only support activation fuctions `Tanh` and `Sigmoid` (currently), which excludes the benefits of more modern activation functions like `Rectifier`.

Customizing Learning

Training Callbacks

You have full access to — and some control over — the internal mechanism of the training algorithm via callback functions. There are six callbacks available:

- `on_train_start` — Called when the main training function is entered.
- `on_epoch_start` — Called the first thing when a new iteration starts.
- `on_batch_start` — Called before an individual batch is processed.
- `on_batch_finish` — Called after that individual batch is processed.
- `on_epoch_finish` — Called the first last when the iteration is done.
- `on_train_finish` — Called just before the training function exits.

You can register for callbacks with a single function, for example:

```
def my_callback(event, **variables):
    print(event)        # The name of the event, as shown in the list above.
    print(variables)     # Full dictionary of local variables from training loop.

nn = Regressor(layers=[Layer("Linear")],
               callback=my_callback)
```

This function will get called for each event, which may be thousands of times depending on your dataset size. An easier way to proceed would be to use specialized callbacks. For example, you can use callbacks on each epoch to mutate or jitter the data for training, or inject new data lazily as it is loaded.

```
def prepare_data(X, y, **other):
    # X and y are variables in the training code. Modify them
    # here to use new data for the next epoch.
    X[:] = X_new
    y[:] = y_new

nn = Regressor(layers=[Layer("Linear")],
               callback={'on_epoch_start': prepare_data})
```

This callback will only get triggered at the start of each epoch, before any of the data in the set has been processed. You can also prepare the data separately in a thread and inject it into the training loop at the last minute.

Epoch Statistics

You can access statistics from the training by using another callback, specifically `on_epoch_finish`. There, multiple variables are accessible including `avg_valid_error` and `avg_train_error` which contain the mean squared error of the last epoch, but you can also access the best results so far via `best_valid_error` and `best_train_error`.

```
errors = []
def store_stats(avg_valid_error, avg_train_error, **_):
    errors.append((avg_valid_error, avg_train_error))

nn = Classifier(
    layers=[Layer("Softmax")], n_iter=5,
    callback={'on_epoch_finish': store_stats})
```

After the training, you can then plot the content of the `errors` variable using your favorite graphing library.

CHAPTER 3

Indices & Search

- genindex
- search

Index