
PyBrain Documentation

Release 0.2

IDSIA/CogBotLab

October 17, 2008

CONTENTS

1 Quickstart	3
1.1 Installing PyBrain and its dependencies	3
1.2 Building a Network	4
1.3 Building a DataSet	6
1.4 Training your Network on your DataSet	7
2 Tutorial	9
2.1 Introduction	9
2.2 Building Networks with Modules and Connections	10
2.3 Classification with Feed-Forward Neural Networks	13
3 API	17
3.1 connections – Structural Components: Connections	17
3.2 modules – Structural Components: Modules	17
3.3 networks – Structural Components: Networks	18
3.4 classification – Datasets for Supervised Classification Training	19
3.5 sequential – Dataset for Supervised Sequences Regression Training	20
3.6 supervised – Dataset for Supervised Regression Training	21
3.7 svmunit – LIBSVM Support Vector Machine Unit	22
3.8 svmtrainer – LIBSVM Support Vector Machine Trainer	22
3.9 tools – Some Useful Tools and Macros	23
3.10 trainers – Supervised Training for Networks and other Modules	26
4 Indices and tables	29
Module Index	31
Index	33

The documentation is build up in the following parts: first, there is the quickstart tutorial which aims at getting you started with PyBrain as quickly as possible. This is the right place for you if you just want get a feel for the library or if you never used PyBrain before.

Although the quickstart uses supervised learning with neural networks as an example, this does not mean that that's it. PyBrain is not only about supervised learning and neural networks.

While the quickstart should be read sequentially, the tutorial's chapters can be read mostly independent of each other.

In case this does not suffice, we also have an API reference, the *Module Index*. Most of the packages and modules of PyBrain are auto-documented there.

If at any point the documentation does not suffice, you can always get help at the pybrain Google Group at <http://groups.google.com/group/pybrain>.

Quickstart

1.1 Installing PyBrain and its dependencies

There are several requirements for PyBrain and we will go through them step by step.

1.1.1 Python

On most Unix systems as Linux or Mac OS X, Python is already installed, but you should check if it is the right version - otherwise you will get weird errors. One way to do this is on the commandline:

```
$ python --version
Python 2.5.2
```

Note: Most command line examples will have a Unix look, but all of them are usable on Windows systems, too.

If you have not installed Python, you should go to the [official Python download page](#) and follow the instructions there.

1.1.2 Setuptools

Setuptools is a package that contains Python's package manager and makes installation of python packages easier. We will use this for the requirements.

First, download this [bootstrap script](#) and run it:

```
$ python ez_setup.py
```

Note: On Unix systems, you will possibly have to run these commands as a superuser.

1.1.3 Scipy and Matplotlib

This will install the python package manager Easy Install for you. Afterwards, you can do:

```
$ easy_install scipy
$ easy_install matplotlib
```

This might take longer, since scipy will also compile different things.

1.1.4 PyBrain

There are two ways of installing PyBrain: you can either go with our stable release on [the webpage](#) or the bleeding edge from our SVN repository.

To check out the PyBrain repository run:

```
$ svn co http://pybrain-svn.informatik.tu-muenchen.de/trunk/ pybrain-trunk
```

Change into the directory Pybrain directory after either checking out SVN or downloading and extracting the archive. There you run (possibly as a superuser on Unix systems):

```
$ python setup.py install
```

Now everything should be installed. To test if it works fine, go into Python and import PyBrain. You should see something like this:

```
$ python
Python 2.5.2 (r252:60911, Sep 17 2008, 11:21:23)
[GCC 4.0.1 (Apple Inc. build 5465)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import pybrain
>>>
```

If this works, you are set.

1.2 Building a Network

To go on with the quickstart, just fire up Python and we will make everything in the interpreter:

```
$ python
Python 2.5.2 (r252:60911, Sep 17 2008, 11:21:23)
[GCC 4.0.1 (Apple Inc. build 5465)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

In PyBrain, networks are composed of Modules which are connected with Connections. You can think of a network as a directed acyclic graph, where the nodes are Modules and the edges are Connections. This makes PyBrain very flexible but it is also not necessary in all cases.

1.2.1 The buildNetwork Shortcut

Thus, there is a simple way to create networks, which is the `buildNetwork` shortcut:

```
>>> from pybrain.tools.shortcuts import buildNetwork
>>> net = buildNetwork(2, 3, 1)
```

This call returns a network that has two inputs, three hidden and a single output neuron. In PyBrain, these layers are Module objects and they are already connected with `FullConnection` objects.

1.2.2 Activating a Network

The net is already initialized with random values - we can already calculate its output:

```
>>> net.activate([2, 1])
array([-0.98646726])
```

For this we use the `.activate()` method, which expects a list, tuple or an array as input.

1.2.3 Examining the structure

How can we examine the structure of our network somewhat closer? In PyBrain, every part of a network has a name by which you can access it. When building networks with the `buildNetwork` shortcut, the parts are named automatically:

```
>>> net['in']
<LinearLayer 'in'>
>>> net['hidden0']
<SigmoidLayer 'hidden0'>
>>> net['out']
<LinearLayer 'out'>
```

The hidden layers have numbers at the end in order to distinguish between those.

1.2.4 More sophisticated Networks

Of course, we want more flexibility when building up networks. For instance, the hidden layer is constructed with the sigmoid squashing function per default: but in a lot of cases, this is not what we want. We can also supply different types of layers:

```
>>> from pybrain.structure import TanhLayer
>>> net = buildNetwork(2, 3, 1, hiddenclass=TanhLayer)
>>> net['hidden0']
<TanhLayer 'hidden0'>
```

There is more we can do. For example, we can also set a different class for the output layer:

```
>>> from pybrain.structure import SoftmaxLayer
>>> net = buildNetwork(2, 3, 2, hiddenclass=TanhLayer, outclass=SoftmaxLayer)
>>> net.activate((2, 3))
array([ 0.6656323,  0.3343677])
```

We can also tell the network to use a bias:

```
>>> net = buildNetwork(2, 3, 1, bias=True)
>>> net['bias']
<BiasUnit 'bias'>
```

This approach has of course some restrictions: for example, we can only construct a feedforward topology. But it is possible to create very sophisticated architectures with PyBrain, and it is also one of the library's strength to do so.

1.3 Building a DataSet

In order for our networks to learn anything, we need a dataset that contains inputs and targets. PyBrain has the `pybrain.dataset` package for this, and we will use the `SupervisedDataSet` class for our needs.

1.3.1 A customized DataSet

The `SupervisedDataSet` class is used for standard supervised learning. It supports input and target values, whose size we have to specify on object creation:

```
>>> from pybrain.datasets import SupervisedDataSet
>>> ds = SupervisedDataSet(2, 1)
```

Here we have generated a dataset that supports two dimensional inputs and one dimensional targets.

1.3.2 Adding samples

A classic example for neural network training is the XOR function, so let's just build a dataset for this. We can do this by just adding samples to the dataset:

```
>>> ds.addSample((0, 0), (0,))
>>> ds.addSample((0, 1), (1,))
>>> ds.addSample((1, 0), (1,))
>>> ds.addSample((1, 1), (0,))
```

1.3.3 Examining the dataset

We now have a dataset that has 4 samples in it. We can check that with python's idiomatic way of checking the size of something:

```
>>> len(ds)
4
```

We can also iterate over it in the standard way:

```
>>> for inpt, target in ds:
...     print inpt, target
...
[ 0.  0.] [ 0.]
[ 0.  1.] [ 1.]
[ 1.  0.] [ 1.]
[ 1.  1.] [ 0.]
```

We can access the input and target field directly as arrays:

```
>>> ds['input']
array([[ 0.,  0.],
       [ 0.,  1.],
       [ 1.,  0.],
       [ 1.,  1.]])
>>> ds['target']
array([[ 0.],
```

```
[ 1.],  
[ 1.],  
[ 0.]])
```

It is also possible to clear a dataset again, and delete all the values from it:

```
>>> ds.clear()  
>>> ds['input']  
array([], shape=(0, 2), dtype=float64)  
>>> ds['target']  
array([], shape=(0, 1), dtype=float64)
```

1.4 Training your Network on your Dataset

For adjusting parameters of modules in supervised learning, PyBrain has the concept of trainers. Trainers take a module and a dataset in order to train the module to fit the data in the dataset.

A classic example for training is backpropagation. PyBrain comes with backpropagation, of course, and we will use the `BackpropTrainer` here:

```
>>> from pybrain.supervised.trainers import BackpropTrainer
```

We have already build a dataset for XOR and we have also learned to build networks that can handle such problems. Let's just connect the two with a trainer:

```
>>> net = buildNetwork(2, 3, 1, bias=True, hiddenclass=TanhLayer)  
>>> trainer = BackpropTrainer(net, ds)
```

The trainer now knows about the network and the dataset and we can train the net on the data:

```
>>> trainer.train()  
0.31516384514375834
```

This call trains the net for one full epoch and returns a double proportional to the error. If we want to train the network until convergence, there is another method:

```
>>> trainer.trainUntilConvergence()  
...
```

This returns a whole bunch of data, which is nothing but a tuple containing the errors for every training epoch.

Tutorial

2.1 Introduction

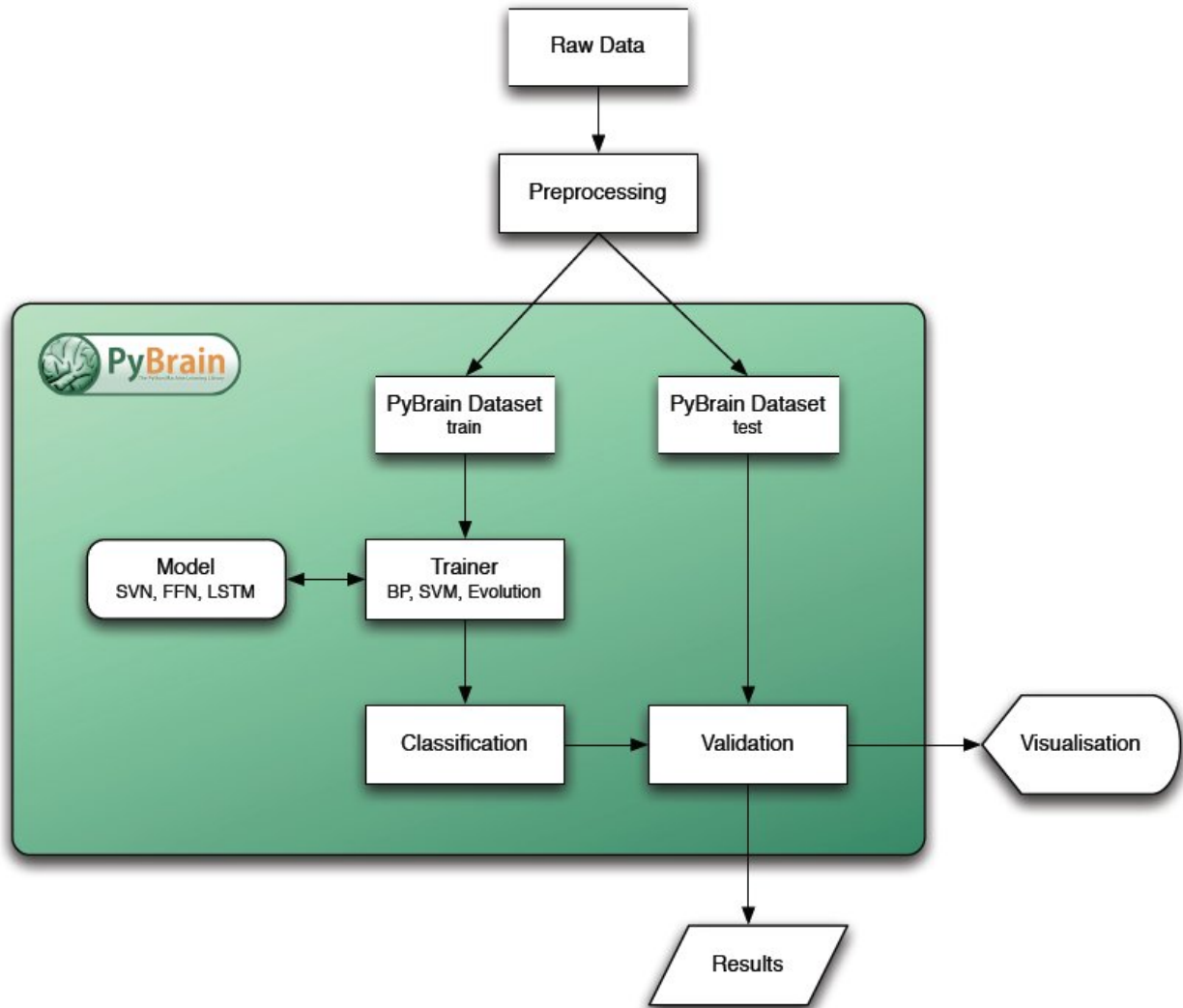
PyBrain's concept is to encapsulate different data processing algorithms in what we call a `Module`. A minimal `Module` contains a forward implementation depending on a collection of free parameters that can be adjusted, usually through some machine learning algorithm.

Modules have an input and an output buffer, plus corresponding error buffers which are used in error backpropagation algorithms.

They are assembled into objects of the class `Network` and are connected via `Connection` objects. These may contain a number of adjustable parameters themselves, such as weights.

Note that a `Network` itself is again a `Module`, such that it is easy to build hierarchical networks as well. Shortcuts exist for building the most common network architectures, but in principle this system allows almost arbitrary connectionist systems to be assembled, as long as they form a directed acyclic graph.

The free parameters of the `Network` are adjusted by means of a `Trainer`, which uses a `Dataset` to learn the optimum parameters from examples. For reinforcement learning experiments, a simulation environment with an associated optimization task is used instead of a `Dataset`.



2.2 Building Networks with Modules and Connections

This chapter will guide you to use PyBrain's most basic structural elements: the `FeedForwardNetwork` and `RecurrentNetwork` classes and with them the `Module` class and the `Connection` class. We have already seen how to create networks with the `buildNetwork` shortcut - but since this technique is limited in some ways, we will now explore how to create networks from the ground up.

2.2.1 Feed Forward Networks

We will start with a simple example, building a multi layer perceptron.

First we make a new `FeedForwardNetwork` object:

```
>>> from pybrain.structure import FeedForwardNetwork
>>> n = FeedForwardNetwork()
```

Next, we're constructing the input, hidden and output layers:

```
>>> from pybrain.structure import LinearLayer, SigmoidLayer
>>> inLayer = LinearLayer(2)
>>> hiddenLayer = SigmoidLayer(3)
>>> outLayer = LinearLayer(1)
```

There are a couple of different classes of layers. For a complete list check out the `modules` package.

In order to use them, we have to add them to the network:

```
>>> n.addInputModule(inLayer)
>>> n.addModule(hiddenLayer)
>>> n.addOutputModule(outLayer)
```

We can actually add multiple input and output modules. The net has to know which of its modules are input and output modules, in order to forward propagate input and to back propagate errors.

It still needs to be explicitly determined how they should be connected. For this we use the most common connection type, which produces a full connectivity between layers, by connecting each neuron of one layer with each neuron of the other. This is implemented by the `FullConnection` class:

```
>>> from pybrain.structure import FullConnection
>>> in_to_hidden = FullConnection(inLayer, hiddenLayer)
>>> hidden_to_out = FullConnection(hiddenLayer, outLayer)
```

As with modules, we have to explicitly add them to the network:

```
>>> n.addConnection(in_to_hidden)
>>> n.addConnection(hidden_to_out)
```

All the elements are in place now, so we can do the final step that makes our MLP usable, which is to call the `.sortModules()` method:

```
>>> n.sortModules()
```

This call does some internal initialization which is necessary before the net can finally be used: for example, the modules are sorted topologically.

2.2.2 Examining a Network

We can actually print networks and examine their structure:

```
>>> print n
FeedForwardNetwork-6
Modules:
  [<LinearLayer 'LinearLayer-3'>, <SigmoidLayer 'SigmoidLayer-7'>, <LinearLayer 'LinearLayer-8'>]
Connections:
  [<FullConnection 'FullConnection-4': 'LinearLayer-3' -> 'SigmoidLayer-7'>, <FullConnection 'FullConn
```

Note that the output on your machine will not necessarily be the same.

One way of using the network is to call its `'activate()'` method with an input to be transformed:

```
>>> n.activate([1, 2])
array([-0.11302355])
```

Again, this might look different on your machine - the weights of the connections have already been initialized randomly. To have a look at those parameters, just check the `.params` field of the connections:

We can access the trainable parameters (weights) of a connection directly, or read all weights of the network at once:

```
>>> in_to_hidden.params
array([ 1.37751406,  1.39320901, -0.24052686, -0.67970042, -0.5999425 , -1.27774679])
>>> hidden_to_out.params
array([-0.32156782,  1.09338421,  0.48784924])
```

The network encapsulating the modules actually holds the parameters too. You can check them out here:

```
>>> n.params
array([ 1.37751406,  1.39320901, -0.24052686, -0.67970042, -0.5999425 ,
       -1.27774679, -0.32156782,  1.09338421,  0.48784924])
```

As you can see, the last three parameters of the network equal the parameters of the second connection.

2.2.3 Naming your Networks structure

In some settings it makes sense to give the parts of a network explicit identifiers. The structural components are derived from the `Named` class, which means that they have an attribute `.name` by which you can identify it by. If no name is given, a new name will be generated automatically.

Subclasses can also be named by passing the `name` argument on initialization:

```
>>> LinearLayer(2)
<LinearLayer 'LinearLayer-11'>
>>> LinearLayer(2, name="foo")
<LinearLayer 'foo'>
```

By using names for your networks, printouts look more concise and readable. They also ensure that your network components are named in the same way every time you run your program.

2.2.4 Using Recurrent Networks

In order to allow recurrency, networks have to be able to “look back in time”. Due to this, the `RecurrentNetwork` class is different from the `FeedForwardNetwork` class in the substantial way, that the complete history is saved. This is actually memory consuming, but necessary for some learning algorithms.

To create a recurrent network, just do as with feedforward networks but use the appropriate class:

```
>>> from pybrain.structure import RecurrentNetwork
>>> n = RecurrentNetwork()
```

We will quickly build up a network that is the same as in the example above:

```
>>> n.addInputModule(LinearLayer(2, name='in'))
>>> n.addModule(SigmoidLayer(3, name='hidden'))
>>> n.addOutputModule(LinearLayer(1, name='out'))
>>> n.addConnection(FullConnection(n['in'], n['hidden'], name='c1'))
>>> n.addConnection(FullConnection(n['hidden'], n['out'], name='c2'))
```

The `RecurrentNetwork` class has one additional method, `.addRecurrentConnection()`, which looks back in time one timestep. We can add one from the hidden to the hidden layer:

```
>>> n.addRecurrentConnection(FullConnection(n['hidden'], n['hidden'], name='c3'))
```

If we now activate the network, we will get different outputs each time:

```
>>> n.sortModules()
>>> n.activate((2, 2))
array([-0.1959887])
>>> n.activate((2, 2))
array([-0.19623716])
>>> n.activate((2, 2))
array([-0.19675801])
```

Of course, we can clear the history of the network. This can be done by calling the *reset* method:

```
>>> n.reset()
>>> n.activate((2, 2))
array([-0.1959887])
>>> n.activate((2, 2))
array([-0.19623716])
>>> n.activate((2, 2))
array([-0.19675801])
```

After the call to `.reset()`, we are getting the same outputs as just after the objects creation.

2.3 Classification with Feed-Forward Neural Networks

This tutorial walks you through the process of setting up a dataset for classification, and train a network on it while visualizing the results online.

First we need to import the necessary components from PyBrain.

```
from pybrain.datasets          import ClassificationDataSet
from pybrain.utilities         import percentError
from pybrain.tools.shortcuts   import buildNetwork
from pybrain.supervised.trainers import BackpropTrainer
from pybrain.structure.modules import SoftmaxLayer
```

Furthermore, pylab is needed for the graphical output.

```
from pylab import ion, ioff, figure, draw, contourf, clf, show, hold, plot
from scipy import diag, arange, meshgrid, where
from numpy.random import multivariate_normal
```

To have a nice dataset for visualization, we produce a set of points in 2D belonging to three different classes. You could also read in your data from a file, e.g. using `pylab.load()`.

```
means = [(-1,0), (2,4), (3,1)]
cov = [diag([1,1]), diag([0.5,1.2]), diag([1.5,0.7])]
alldata = ClassificationDataSet(2, 1, nb_classes=3)
for n in xrange(400):
    for klass in range(3):
        input = multivariate_normal(means[klass], cov[klass])
        alldata.addSample(input, [klass])
```

Randomly split the dataset into 75% training and 25% test data sets. Of course, we could also have created two different datasets to begin with.

```
tstdata, trndata = alldata.splitWithProportion( 0.25 )
```

For neural network classification, it is highly advisable to encode classes with one output neuron per class. Note that this operation duplicates the original targets and stores them in an (integer) field named 'class'.

```
trndata._convertToOneOfMany( )
tstdata._convertToOneOfMany( )
```

Test our dataset by printing a little information about it.

```
print "Number of training patterns: ", len(trndata)
print "Input and output dimensions: ", trndata.indim, trndata.outdim
print "First sample (input, target, class):"
print trndata['input'][0], trndata['target'][0], trndata['class'][0]
```

Now build a feed-forward network with 5 hidden units. We use the shortcut `buildNetwork()` for this. The input and output layer size must match the dataset's input and target dimension. You could add additional hidden layers by inserting more numbers giving the desired layer sizes.

The output layer uses a softmax function because we are doing classification. There are more options to explore here, e.g. try changing the hidden layer transfer function to linear instead of (the default) sigmoid.

See Also:

Description `buildNetwork()` for more info on options, and the Network tutorial *[Building Networks with Modules and Connections](#)* for info on how to build your own non-standard networks.

```
fnn = buildNetwork( trndata.indim, 5, trndata.outdim, outclass=SoftmaxLayer )
```

Set up a trainer that basically takes the network and training dataset as input. For a list of trainers, see `trainers`. We are using a `BackpropTrainer` for this.

```
trainer = BackpropTrainer( fnn, dataset=trndata, momentum=0.1, verbose=True, weightdecay=0.01)
```

Now generate a square grid of data points and put it into a dataset, which we can then classify to obtain a nice contour field for visualization. Therefore the target values for this data set can be ignored.

```
ticks = arange(-3.,6.,0.2)
X, Y = meshgrid(ticks, ticks)
# need column vectors in dataset, not arrays
griddata = ClassificationDataSet(2,1, nb_classes=3)
for i in xrange(X.size):
    griddata.addSample([X.ravel()[i],Y.ravel()[i]], [0])
griddata._convertToOneOfMany() # this is still needed to make the fnn feel comfy
```

Start the training iterations.

```
for i in range(20):
```

Train the network for some epochs. Usually you would set something like 5 here, but for visualization purposes we do this one epoch at a time.

```
...
    trainer.trainEpochs( 1 )
```

Evaluate the network on the training and test data. There are several ways to do this - check out the `pybrain.tools.validation` module, for instance. Here we let the trainer do the test.

```
...
    trnresult = percentError( trainer.testOnClassData(),
                             trndata['class'] )
    tstresult = percentError( trainer.testOnClassData(
                             dataset=tstdata ), tstdata['class'] )

    print "epoch: %4d" % trainer.totalepochs, \
          "  train error: %5.2f%%" % trnresult, \
          "  test error: %5.2f%%" % tstresult
```

Run our grid data through the FNN, get the most likely class and shape it into a square array again.

```
...
    out = fnn.activateOnDataset(griddata)
    out = out.argmax(axis=1) # the highest output activation gives the class
    out = out.reshape(X.shape)
```

Now plot the test data and the underlying grid as a filled contour.

```
...
    figure(1)
    ioff() # interactive graphics off
    clf()  # clear the plot
    hold(True) # overplot on
    for c in [0,1,2]:
        here, _ = where(tstdata['class']==c)
        plot(tstdata['input'][here,0],tstdata['input'][here,1], 'o')
    if out.max()!=out.min(): # safety check against flat field
        contourf(X, Y, out) # plot the contour
    ion() # interactive graphics on
    draw() # update the plot
```

Finally, keep showing the plot until user kills it.

```
ioff()
show()
```


API

toctree glob pattern u'api/supervised/*' didn't match any documents

3.1 connections – Structural Components: Connections

class FullConnection (**args, **kwargs*)

Connection which fully connects every element from the first module's output buffer to the second module's input buffer in a matrix multiplicative manner.

whichBuffers (*paramIndex*)

Return the index of the input module's output buffer and the output module's input buffer for the given weight.

class IdentityConnection (**args, **kwargs*)

Connection which connects the *i*'th element from the first module's output buffer to the *i*'th element of the second module's input buffer.

class SharedConnection (*mother, *args, **kwargs*)

A shared connection can link different couples of modules, with a single set of parameters.

class SharedFullConnection (*mother, *args, **kwargs*)

Shared version of FullConnection.

class MotherConnection (*nbparams, **args*)

The container for the shared parameters of connections (just a container with a constructor, actually).

3.2 modules – Structural Components: Modules

class BiasUnit (*name=None*)

Bases: `pybrain.structure.modules.neuronlayer.NeuronLayer`

A simple bias unit with a single constant output.

class GaussianLayer (*dim, name=None*)

Bases: `pybrain.structure.modules.neuronlayer.NeuronLayer`,
`pybrain.structure.parametercontainer.ParameterContainer`

A layer implementing a gaussian interpretation of the input. The mean is the input, the sigmas are stored in the module parameters.

setSigma (*sigma*)

Wrapper method to set the sigmas (the parameters of the module) to a certain value.

class LinearLayer (*dim, name=None*)

Bases: `pybrain.structure.modules.neuronlayer.NeuronLayer`

The simplest kind of module, not doing any transformation.

__init__ (*dim, name=None*)

Create a layer with *dim* number of units.

class LSTMLayer (*dim, peepholes=False, name=None*)

Bases: `pybrain.structure.modules.neuronlayer.NeuronLayer`,
`pybrain.structure.parametercontainer.ParameterContainer`

Long short-term memory cell layer.

The input consists of 4 parts, in the following order: - input gate - forget gate - cell input - output gate

class MDLSTMLayer (*dim, dimensions=1, peepholes=False, name=None*)

Bases: `pybrain.structure.modules.neuronlayer.NeuronLayer`,
`pybrain.structure.parametercontainer.ParameterContainer`

Multi-dimensional long short-term memory cell layer.

The cell-states are explicitly passed on through a part of the input/output buffers (which should be connected correctly with IdentityConnections).

The input consists of 4 parts, in the following order: - input gate - forget gates (1 per dim) - cell input - output gate - previous states (1 per dim)

The output consists of two parts: - cell output - current state

Attention: this module has to be used with care: it's last <size> input and outputs are reserved for transmitting internal states on flattened recursive multi-dim networks, and so its connections have always to be sliced!

class SigmoidLayer (*dim, name=None*)

Bases: `pybrain.structure.modules.neuronlayer.NeuronLayer`

Layer implementing the sigmoid squashing function.

__init__ (*dim, name=None*)

Create a layer with *dim* number of units.

class SoftmaxLayer (*dim, name=None*)

Bases: `pybrain.structure.modules.neuronlayer.NeuronLayer`

A layer implementing a softmax distribution over the input.

__init__ (*dim, name=None*)

Create a layer with *dim* number of units.

class StateDependentLayer (*dim, module, name=None, onesigma=True*)

Bases: `pybrain.structure.modules.neuronlayer.NeuronLayer`,
`pybrain.structure.parametercontainer.ParameterContainer`

class TanhLayer (*dim, name=None*)

Bases: `pybrain.structure.modules.neuronlayer.NeuronLayer`

A layer implementing the tanh squashing function.

__init__ (*dim, name=None*)

Create a layer with *dim* number of units.

3.3 networks – Structural Components: Networks

class Network (*name=None, **args*)

Bases: `pybrain.structure.modules.module.Module`, `pybrain.structure.parametercontainer.ParameterContainer`

Abstract class for linking different modules with connections.

activate (*inpt*)
Do one transformation of an input and return the result.

activateOnDataset (*dataset*)
Run the module's forward pass on the given dataset unconditionally and return the output.

addConnection (*c*)
Add the given connection to the network.

addInputModule (*m*)
Add the given module to the network and mark it as an input module.

addModule (*m*)
Add the given module to the network.

addOutputModule (*m*)
Add the given module to the network and mark it as an output module.

reset ()
Reset all component modules and the network.

sortModules ()
Prepare the network for activation by sorting the internal datastructure.
Needs to be called before activation.

class FeedForwardNetwork (**args, **kwargs*)
Bases: `pybrain.structure.networks.feedforward.FeedForwardNetworkComponent`,
`pybrain.structure.networks.network.Network`
FeedForwardNetworks are networks that do not work for sequential data. Every input is treated as independent of any previous or following inputs.

class RecurrentNetwork (**args, **kwargs*)
Bases: `pybrain.structure.networks.recurrent.RecurrentNetworkComponent`,
`pybrain.structure.networks.network.Network`
Class that implements networks which can work with sequential data.
Until `.reset()` is called, the network keeps track of all previous inputs and thus allows the use of recurrent connections and layers that look back in time.

addRecurrentConnection (*c*)
Add a connection to the network and mark it as a recurrent one.

3.4 classification – Datasets for Supervised Classification Training

class ClassificationDataSet (*inp, target=1, nb_classes=0, class_labels=None*)
Bases: `pybrain.datasets.supervised.SupervisedDataSet`
Specialized data set for classification data. Classes are to be numbered from 0 to `nb_classes-1`.

__init__ (*inp, target=1, nb_classes=0, class_labels=None*)
Initialize an empty dataset.
inp is used to specify the dimensionality of the input. While the number of targets is given by implicitly by the training samples, it can also be set explicitly by *nb_classes*. To give the classes names, supply an iterable of strings as *class_labels*.

calculateStatistics ()
Return a class histogram.

getClass (*idx*)
Return the label of given class.

splitByClass (*cls_select*)

Produce two new datasets, the first one comprising only the class selected (0..nClasses-1), the second one containing the remaining samples.

castToRegression (*values*)

Converts data set into a SupervisedDataSet for regression. Classes are used as indices into the value array given.

_convertToOneOfMany (*bounds*=(0, 1))

Converts the target classes to a 1-of-k representation, retaining the old targets as a field *class*.

To supply specific bounds, set the *bounds* parameter, which consists of target values for non-membership and membership.

_convertToClassNb ()

The reverse of **_convertToOneOfMany**. Target field is overwritten.

class SequenceClassificationDataSet (*inp, target, nb_classes=0, class_labels=None*)

Bases: `pybrain.datasets.sequential.SequentialDataSet`,
`pybrain.datasets.classification.ClassificationDataSet`

Defines a dataset for sequence classification. Each sample in the sequence still needs its own target value.

__init__ (*inp, target, nb_classes=0, class_labels=None*)

Initialize an empty dataset.

inp is used to specify the dimensionality of the input. While the number of targets is given by implicitly by the training samples, it can also be set explicitly by *nb_classes*. To give the classes names, supply an iterable of strings as *class_labels*.

3.5 sequential – Dataset for Supervised Sequences Regression Training

class SequentialDataSet (*indim, targetdim*)

Bases: `pybrain.datasets.supervised.SupervisedDataSet`

A SequentialDataSet is like a SupervisedDataSet except that it can keep track of sequences of samples. Indices of a new sequence are stored whenever the method newSequence() is called. The last (open) sequence is considered a normal sequence even though it does not have a following “new sequence” marker.

endOfSequence (*index*)

Return True if the marker was moved over the last element of sequence *index*, False otherwise.

Mostly used like .endOfData() with while loops.

evaluateModuleMSE (*module, averageOver=1, **args*)

Evaluate the predictions of a module on a sequential dataset and return the MSE (potentially average over a number of epochs).

getCurrentSequence ()

Return the current sequence, according to the marker position.

getNumSequences ()

Return the number of sequences. The last (open) sequence is also counted in, even though there is no additional ‘newSequence’ marker.

getSequence (*index*)

Returns the sequence given by *index*.

A list of arrays is returned for the linked arrays. It is assumed that the last sequence goes until the end of the dataset.

getSequenceIterator (*index*)

Return an iterator over the samples of the sequence specified by *index*.

getSequenceLength (*index*)

Return the length of the given sequence. If *index* is pointing to the last sequence, the sequence is considered to go until the end of the dataset.

gotoSequence (*index*)

Move the internal marker to the beginning of sequence *index*.

newSequence ()

Marks the beginning of a new sequence. this function does nothing if called at the very start of the data set. Otherwise, it starts a new sequence. Empty sequences are not allowed, and an `EmptySequenceError` exception will be raised.

removeSequence (*index*)

Remove the *index*'th sequence from the dataset and places the marker to the sample following the removed sequence.

splitWithProportion (*proportion=0.5*)

Produce two new datasets, each containing a part of the sequences.

The first dataset will have a fraction given by *proportion* of the dataset.

Note: This documentation comprises just a subjective excerpt of available methods. See the source code for additional functionality.

3.6 supervised – Dataset for Supervised Regression Training

class SupervisedDataSet (*inp, target*)

Bases: `pybrain.datasets.dataset.DataSet`

SupervisedDataSets have two fields, one for input and one for the target.

__init__ (*inp, target*)

Initialize an empty supervised dataset.

Pass *inp* and *target* to specify the dimensions of the input and target vectors.

__len__ ()

Return the length of the linked data fields. If no linked fields exist, return the length of the longest field.

addSample (*inp, target*)

Add a new sample consisting of *input* and *target*.

batches (*label, n, permutation=None*)

Yield batches of the size of *n* from the dataset.

A single batch is an array of with *dim* columns and *n* rows. The last batch is possibly smaller.

If *permutation* is given, batches are yielded in the corresponding order.

clear (*unlinked=False*)

Clear the dataset.

If linked fields exist, only the linked fields will be deleted unless *unlinked* is set to `True`. If no fields are linked, all data will be deleted.

copy ()

Return a deep copy.

static **loadFromFile** (*filename, format=None*)

Return an instance of the class that is saved in the file with the given filename in the specified format.

randomBatches (*label, n*)

Like `.batches()`, but the order is random.

static **reconstruct** (*filename*)

Read an incomplete data set (option arrayonly) into the given one.

saveToFile (*filename*, *format=None*, ***kwargs*)

Save the object to file given by filename.

splitWithProportion (*proportion=0.5*)

Produce two new datasets, the first one containing the fraction given by *proportion* of the samples.

3.7 svmunit – LIBSVM Support Vector Machine Unit

class SVMUnit (*indim=0*, *outdim=0*, *model=None*)

This unit represents an Support Vector Machine and is implemented through the LIBSVM Python interface. It functions somewhat like a Model or a Network, but combining it with other PyBrain Models is currently discouraged. Its main function is to compare against feed-forward network classifications. You cannot get or set model parameters, but you can load and save the entire model in LIBSVM format. Sequential data and backward passes are not supported. See the corresponding example code for usage.

__init__ (*indim=0*, *outdim=0*, *model=None*)

Initializes as empty module.

If *model* is given, initialize using this LIBSVM model instead. *indim* and *outdim* are for compatibility only currently, and ignored.

forwardPass (*values=False*)

Produce the output from the current input vector, or process a dataset.

If *values* is False or 'class', output is set to the number of the predicted class. If True or 'raw', produces decision values instead. These are stored in a dictionary for multi-class SVM. If *prob*, class probabilities are produced. This works only if probability option was set for SVM training.

activateOnDataset (*dataset*, *values=False*)

Run the module's forward pass on the given dataset unconditionally and return the output as a list. @param dataset: A non-sequential supervised data set. @param values: Passed trough to forwardPass() method.

getNbClasses ()

return number of classes the current model uses

reset ()

Reset input and output buffers

setModel (*model*)

Set the SVM model.

loadModel (*filename*)

Read the SVM model description from a file

saveModel (*filename*)

Save the SVM model description from a file

3.8 svmtrainer – LIBSVM Support Vector Machine Trainer

class SVMTrainer (*svmunit*, *dataset*, *modelfile=None*, *plot=False*)

A class performing supervised learning of a DataSet by an SVM unit. See the remarks on SVMUnit above. This whole class is a bit of a hack, and provided mostly for convenience of comparisons.

__init__ (*svmunit*, *dataset*, *modelfile=None*, *plot=False*)

Initialize data and unit to be trained, and load the model, if provided.

The passed *svmunit* has to be an object of class SVMUnit that is going to be trained on the ClassificationDataSet object dataset. Compared to FNN training we do not use a test data set, instead 5-fold cross-validation is performed if needed.

If *modelfile* is provided, this model is loaded instead of training. If *plot* is True, a grid search is performed and the resulting pattern is plotted.

train (*search=False, **kwargs*)

Train the SVM on the dataset. For RBF kernels (the default), an optional meta-parameter search can be performed. @param search: optional name of grid search class to use for RBF kernels: 'GridSearch' or 'GridSearchDOE' @param log2g: base 2 log of the RBF width parameter @param log2C: base 2 log of the slack parameter @param searchlog: filename into which to dump the search log @param others: ...are passed through to the grid search and/or libsvm

setParams (***kwargs*)

Set parameters for SVM training. Apart from the ones below, you can use all parameters defined for the LIBSVM svm_model class, see their documentation. @param searchlog: Save a list of coordinates and the achieved CV accuracy to this file.

load (*filename*)

no training at all - just load the SVM model from a file

save (*filename*)

save the trained SVM

class GridSearch (*problem, targets, cmin, cmax, cstep=None, crossval=5, plotflag=False, maxdepth=8, searchlog='gridsearch_results.txt', **params*)

Helper class used by [SVMTrainer](#) to perform an exhaustive grid search, and plot the resulting accuracy surface, if desired. Adapted from the LIBSVM python toolkit.

__init__ (*problem, targets, cmin, cmax, cstep=None, crossval=5, plotflag=False, maxdepth=8, searchlog='gridsearch_results.txt', **params*)

Set up (log) grid search over the two RBF kernel parameters C and gamma. @param problem: the LIBSVM svm_problem to be optimized, ie. the input and target data @param targets: unfortunately, the targets used in the problem definition have to be given again here @param cmin: lower left corner of the log2C/log2gamma window to search @param cmax: upper right corner of the log2C/log2gamma window to search @param cstep: step width for log2C and log2gamma (ignored for DOE search) @param crossval: split dataset into this many parts for cross-validation @param plotflag: if True, plot the error surface contour (regular) or search pattern (DOE) @param maxdepth: maximum window bisection depth (DOE only) @param searchlog: Save a list of coordinates and the achieved CV accuracy to this file @param others: ...are passed through to the cross_validation method of LIBSVM

search ()

iterate successive parameter grid refinement and evaluation; adapted from LIBSVM grid search tool

setParams (***kwargs*)

set parameters for SVM training

class GridSearchDOE (*problem, targets, cmin, cmax, cstep=None, crossval=5, plotflag=False, maxdepth=8, searchlog='gridsearch_results.txt', **params*)

Same as GridSearch, but implements a design-of-experiments based search pattern, as described by C. Staelin, <http://www.hpl.hp.com/techreports/2002/HPL-2002-354R1.pdf>

search (*cmin=None, cmax=None*)

iterate parameter grid refinement and evaluation recursively

3.9 tools – Some Useful Tools and Macros

Neural network tools

buildNetwork (**layers, **options*)

Build arbitrary deep networks.

layers should be a list or tuple of integers, that indicate how many neurons the layers should have. *bias* and *outputbias* are flags to indicate whether the network should have the corresponding biases; both default to True.

To adjust the classes for the layers use the *hiddenclass* and *outclass* parameters, which expect a subclass of *NeuronLayer*.

If the *recurrent* flag is set, a *RecurrentNetwork* will be created, otherwise a *FeedForwardNetwork*.

If the *fast* flag is set, faster arac networks will be used instead of the pybrain implementations.

class NNregression (*DS*, ***kwargs*)

Learns to numerically predict the targets of a set of data, with optional online progress plots.

__init__ (*DS*, ***kwargs*)

Initialize with the training data set *DS*. All keywords given are set as member variables. The following are particularly important: @param *hidden*: number of hidden units @param *TDS*: test data set for checking convergence @param *VDS*: validation data set for final performance evaluation @param *epoinc*: number of epochs to train for, before checking convergence (default: 5)

initGraphics (*ymax=10*, *xmax=-1*)

initialize the interactive graphics output window, and return a handle to the plot

setupNN (*trainer=<class 'pybrain.supervised.trainers.rprop.RPropMinusTrainer'>*, *hidden=None*, ***trnargs*)

Constructs a 3-layer FNN for regression. Optional arguments are passed on to the Trainer class.

runTraining (*convergence=0*, ***kwargs*)

Trains the network on the stored dataset. If convergence is >0, check after that many epoch increments whether test error is going down again, and stop training accordingly. CAVEAT: No support for Sequential datasets!

saveTrainingCurve (*learnfname*)

save the training curves into a file with the given name (CSV format)

saveNetwork (*fname*)

save the trained network to a file

class NNclassifier (*DS*, ***kwargs*)

Learns to classify a set of data, with optional online progress plots.

__init__ (*DS*, ***kwargs*)

Initialize the classifier: the least we need is the dataset to be classified. All keywords given are set as member variables.

initGraphics (*ymax=10*, *xmax=-1*)

initialize the interactive graphics output window, and return a handle to the plot

setupNN (*trainer=<class 'pybrain.supervised.trainers.rprop.RPropMinusTrainer'>*, *hidden=None*, ***trnargs*)

Setup FNN and trainer for classification.

setupRNN (*trainer=<class 'pybrain.supervised.trainers.backprop.BackpropTrainer'>*, *hidden=None*, ***trnargs*)

Setup an LSTM RNN and trainer for sequence classification.

runTraining (*convergence=0*, ***kwargs*)

Trains the network on the stored dataset. If convergence is >0, check after that many epoch increments whether test error is going down again, and stop training accordingly.

saveTrainingCurve (*learnfname*)

save the training curves into a file with the given name (CSV format)

saveNetwork (*fname*)

save the trained network to a file

Dataset tools

convertSequenceToTimeWindows (*DSseq*, *NewClass*, *winsize*)

Converts a sequential classification dataset into time windows of fixed length. Assumes the correct class is given at the last timestep of each sequence. Incomplete windows at the sequence end are pruned. No overlap between

windows. @param DSseq: the sequential data set to cut up @param winsize: size of the data window @param NewClass: class of the windowed data set to be returned (gets initialised with indim*winsize, outdim)

Training performance validation tools

class Validator ()

This class provides methods for the validation of calculated output values compared to their destined target values. It does not know anything about modules or other pybrain stuff. It just works on arrays, hence contains just the core calculations.

The class has just classmethods, as it is used as kind of namespace instead of an object definition.

ESS

Returns the explained sum of squares (ESS). @param output: array of output values @param target: array of target values

MSE

Returns the mean squared error. The multidimensional arrays will get flattened in order to compare them. @param output: array of output values @param target: array of target values @param importance: each squared error will be multiplied with its

Unexpected indentation.

corresponding importance value. After summing up these values, the result will be divided by the sum of all importance values for normalization purposes.

classificationPerformance

Returns the hit rate of the outputs compared to the targets. @param output: array of output values @param target: array of target values

class ModuleValidator ()

This class provides methods for the validation of calculated output values compared to their destined target values. It especially handles pybrains modules and dataset classes. For the core calculations, the Validator class is used.

The class has just classmethods, as it is used as kind of namespace instead of an object definition.

MSE

Returns the mean squared error. @param module: Object of any subclass of pybrain's Module type @param dataset: Dataset object at least containing the fields

Unexpected indentation.

'input' and 'target' (for example SupervisedDataSet)

calculateModuleOutput

Calculates the module's output on the dataset. Can be called with any type of dataset. @param dataset: Any Dataset object containing an 'input' field.

classificationPerformance

Returns the hit rate of the module's output compared to the targets stored inside dataset. @param module: Object of any subclass of pybrain's Module type @param dataset: Dataset object at least containing the fields

Unexpected indentation.

'input' and 'target' (for example SupervisedDataSet)

validate

Abstract validate function, that is heavily used by this class. First, it calculates the module's output on the dataset. In advance, it compares the output to the target values of the dataset through the valfunc function and returns the result.

@param valfunc: A function expecting arrays for output, target and importance (optional). See Validator.MSE for an example.

Definition list ends without a blank line; unexpected unindent.

@param module: Object of any subclass of pybrain's Module type @param dataset: Dataset object at least containing the fields

Unexpected indentation.

'input' and 'target' (for example SupervisedDataSet)

class CrossValidator (*trainer, dataset, n_folds=5, valfunc=<bound method type.classificationPerformance of <class 'pybrain.tools.validation.ModuleValidator'>>, **kwargs*)

Class for crossvalidating data. An object of CrossValidator must be supplied with a trainer that contains a module and a dataset. Then the dataset is shuffled and split up into *n* parts of equal length.

A clone of the trainer and its module is made, and trained with *n*-1 parts of the split dataset. After training, the module is validated with the *n*'th part of the dataset that was not used during training.

This is done for each possible combination of *n*-1 dataset pieces. The mean of the calculated validation results will be returned.

setArgs (***kwargs*)

Set the specified member variables. @param max_epochs: maximum number of epochs the trainer should train the module for. @param verbosity: set verbosity level

validate ()

The main method of this class. It runs the crossvalidation process and returns the validation result (e.g. performance).

testOnSequenceData (*module, dataset*)

Fetch targets and calculate the modules output on dataset. Output and target are in one-of-many format. The class for each sequence is determined by first summing the probabilities for each individual sample over the sequence, and then finding its maximum.

3.10 trainers – Supervised Training for Networks and other Modules

class BackpropTrainer (*module, dataset=None, learningrate=0.01, lrdecay=1.0, momentum=0.0, verbose=False, batchlearning=False, weightdecay=0.0*)

Trainer that trains the parameters of a module according to a supervised dataset (potentially sequential) by backpropagating the errors (through time).

__init__ (*module, dataset=None, learningrate=0.01, lrdecay=1.0, momentum=0.0, verbose=False, batchlearning=False, weightdecay=0.0*)

Create a BackpropTrainer to train the specified *module* on the specified *dataset*.

The learning rate gives the ratio of which parameters are changed into the direction of the gradient. The learning rate decreases by *lrdecay*, which is used to multiply the learning rate after each training step. The parameters are also adjusted with respect to *momentum*, which is the ratio by which the gradient of the last timestep is used.

If *batchlearning* is set, the parameters are updated only at the end of each epoch. Default is False.

weightdecay corresponds to the weightdecay rate, where 0 is no weight decay at all.

setData (*dataset*)

Associate the given dataset with the trainer.

testOnClassData (*dataset=None, verbose=False, return_targets=False*)

Return winner-takes-all classification output on a given dataset.

If no dataset is given, the dataset passed during Trainer initialization is used. If *return_targets* is set, also return corresponding target classes.

train ()

Train the associated module for one epoch.

trainEpochs (*epochs=1, *args, **kwargs*)

Train on the current dataset for the given number of *epochs*.

Additional arguments are passed on to the train method.

trainOnDataset (*dataset, *args, **kwargs*)

Set the dataset and train.

Additional arguments are passed on to the train method.

trainUntilConvergence (*dataset=None, maxEpochs=None, verbose=None, continueEpochs=10, validationProportion=0.25*)

Train the module on the dataset until it converges.

Return the module with the parameters that gave the minimal validation error.

If no dataset is given, the dataset passed during Trainer initialization is used. *validationProportion* is the ratio of the dataset that is used for the validation dataset.

If *maxEpochs* is given, at most that many epochs are trained. Each time validation error hits a minimum, try for *continueEpochs* epochs to find a better one.

Note: This documentation comprises just a subjective excerpt of available methods. See the source code for additional functionality.

class RPropMinusTrainer (*module, etaminus=0.5, etaplus=1.2, deltamin=9.999999999999995e-07, deltamax=5.0, delta0=0.10000000000000001, **kwargs*)

Train the parameters of a module according to a supervised dataset (possibly sequential) by RProp without weight backtracking (aka RProp-, cf. [Igel&Huesken, Neurocomputing 50, 2003]) and without ponderation, ie. all training samples have the same weight.

__init__ (*module, etaminus=0.5, etaplus=1.2, deltamin=9.999999999999995e-07, deltamax=5.0, delta0=0.10000000000000001, **kwargs*)

Set up training algorithm parameters, and objects associated with the trainer. @param module: the module whose parameters should be trained. @param etaminus: factor by which step width is decreased when overstepping (0.5) @param etaplus: factor by which step width is increased when following gradient (1.2) @param delta: step width for each weight @param deltamin: minimum step width (1e-6) @param deltamax: maximum step width (5.0) @param delta0: initial step width (0.1)

Note: See the documentation of [BackpropTrainer](#) for inherited methods.

Indices and tables

- *Index*
- *Module Index*
- *Search Page*

MODULE INDEX

P

- `pybrain.datasets.classification`, [19](#)
- `pybrain.datasets.sequential`, [20](#)
- `pybrain.datasets.supervised`, [21](#)
- `pybrain.structure.connections`, [17](#)
- `pybrain.structure.modules`, [17](#)
- `pybrain.structure.modules.svmunit`, [22](#)
- `pybrain.structure.networks`, [18](#)
- `pybrain.supervised.trainers`, [26](#)
- `pybrain.supervised.trainers.svmtrainer`,
[22](#)
- `pybrain.tools.datasettools`, [24](#)
- `pybrain.tools.neuralnets`, [24](#)
- `pybrain.tools.shortcuts`, [23](#)
- `pybrain.tools.validation`, [25](#)

INDEX

Symbols

`__init__()` (BackpropTrainer method), 26
`__init__()` (ClassificationDataSet method), 19
`__init__()` (GridSearch method), 23
`__init__()` (LinearLayer method), 18
`__init__()` (NNclassifier method), 24
`__init__()` (NNregression method), 24
`__init__()` (RPropMinusTrainer method), 27
`__init__()` (SVMTrainer method), 22
`__init__()` (SVMUnit method), 22
`__init__()` (SequenceClassificationDataSet method), 20
`__init__()` (SigmoidLayer method), 18
`__init__()` (SoftmaxLayer method), 18
`__init__()` (SupervisedDataSet method), 21
`__init__()` (TanhLayer method), 18
`__len__()` (SupervisedDataSet method), 21
`_convertToClassNb()` (ClassificationDataSet method), 20
`_convertToOneOfMany()` (ClassificationDataSet method), 20

A

`activate()` (Network method), 19
`activateOnDataSet()` (Network method), 19
`activateOnDataSet()` (SVMUnit method), 22
`addConnection()` (Network method), 19
`addInputModule()` (Network method), 19
`addModule()` (Network method), 19
`addOutputModule()` (Network method), 19
`addRecurrentConnection()` (RecurrentNetwork method), 19
`addSample()` (SupervisedDataSet method), 21

B

BackpropTrainer (class in pybrain.supervised.trainers), 26
`batches()` (SupervisedDataSet method), 21
BiasUnit (class in pybrain.structure.modules), 17
`buildNetwork()` (in module pybrain.tools.shortcuts), 23

C

`calculateModuleOutput` (ModuleValidator attribute), 25
`calculateStatistics()` (ClassificationDataSet method), 19
`castToRegression()` (ClassificationDataSet method), 20
ClassificationDataSet (class in pybrain.datasets.classification), 19
`classificationPerformance` (ModuleValidator attribute), 25
`classificationPerformance` (Validator attribute), 25
`clear()` (SupervisedDataSet method), 21
`convertSequenceToTimeWindows()` (in module pybrain.tools.datasettools), 24
`copy()` (SupervisedDataSet method), 21
CrossValidator (class in pybrain.tools.validation), 26

E

`endOfSequence()` (SequentialDataSet method), 20
ESS (Validator attribute), 25
`evaluateModuleMSE()` (SequentialDataSet method), 20

F

FeedForwardNetwork (class in pybrain.structure.networks), 19
`forwardPass()` (SVMUnit method), 22
FullConnection (class in pybrain.structure.connections), 17

G

GaussianLayer (class in pybrain.structure.modules), 17
`getClass()` (ClassificationDataSet method), 19
`getCurrentSequence()` (SequentialDataSet method), 20
`getNbClasses()` (SVMUnit method), 22

getNumSequences() (SequentialDataSet method), 20
 getSequence() (SequentialDataSet method), 20
 getSequenceIterator() (SequentialDataSet method), 20
 getSequenceLength() (SequentialDataSet method), 21
 gotoSequence() (SequentialDataSet method), 21
 GridSearch (class in py-brain.supervised.trainers.svmtrainer), 23
 GridSearchDOE (class in py-brain.supervised.trainers.svmtrainer), 23

I

IdentityConnection (class in py-brain.structure.connections), 17
 initGraphics() (NNclassifier method), 24
 initGraphics() (NNregression method), 24

L

LinearLayer (class in pybrain.structure.modules), 18
 load() (SVMTrainer method), 23
 loadFromFile() (SupervisedDataSet static method), 21
 loadModel() (SVMUnit method), 22
 LSTMLayer (class in pybrain.structure.modules), 18

M

MDLSTMLayer (class in pybrain.structure.modules), 18
 ModuleValidator (class in pybrain.tools.validation), 25
 MotherConnection (class in py-brain.structure.connections), 17
 MSE (ModuleValidator attribute), 25
 MSE (Validator attribute), 25

N

Network (class in pybrain.structure.networks), 18
 newSequence() (SequentialDataSet method), 21
 NNclassifier (class in pybrain.tools.neuralnets), 24
 NNregression (class in pybrain.tools.neuralnets), 24

P

pybrain.datasets.classification (module), 19
 pybrain.datasets.sequential (module), 20
 pybrain.datasets.supervised (module), 21
 pybrain.structure.connections (module), 17
 pybrain.structure.modules (module), 17
 pybrain.structure.modules.svmunit (module), 22
 pybrain.structure.networks (module), 18
 pybrain.supervised.trainers (module), 26

pybrain.supervised.trainers.svmtrainer (module), 22
 pybrain.tools.datasettools (module), 24
 pybrain.tools.neuralnets (module), 24
 pybrain.tools.shortcuts (module), 23
 pybrain.tools.validation (module), 25

R

randomBatches() (SupervisedDataSet method), 21
 reconstruct() (SupervisedDataSet static method), 21
 RecurrentNetwork (class in py-brain.structure.networks), 19
 removeSequence() (SequentialDataSet method), 21
 reset() (Network method), 19
 reset() (SVMUnit method), 22
 RPropMinusTrainer (class in py-brain.supervised.trainers), 27
 runTraining() (NNclassifier method), 24
 runTraining() (NNregression method), 24

S

save() (SVMTrainer method), 23
 saveModel() (SVMUnit method), 22
 saveNetwork() (NNclassifier method), 24
 saveNetwork() (NNregression method), 24
 saveToFile() (SupervisedDataSet method), 22
 saveTrainingCurve() (NNclassifier method), 24
 saveTrainingCurve() (NNregression method), 24
 search() (GridSearchDOE method), 23
 search() (GridSearch method), 23
 SequenceClassificationDataSet (class in py-brain.datasets.classification), 20
 SequentialDataSet (class in py-brain.datasets.sequential), 20
 setArgs() (CrossValidator method), 26
 setData() (BackpropTrainer method), 26
 setModel() (SVMUnit method), 22
 setParams() (GridSearch method), 23
 setParams() (SVMTrainer method), 23
 setSigma() (GaussianLayer method), 17
 setupNN() (NNclassifier method), 24
 setupNN() (NNregression method), 24
 setupRNN() (NNclassifier method), 24
 SharedConnection (class in py-brain.structure.connections), 17
 SharedFullConnection (class in py-brain.structure.connections), 17
 SigmoidLayer (class in pybrain.structure.modules), 18
 SoftmaxLayer (class in pybrain.structure.modules), 18
 sortModules() (Network method), 19
 splitByClass() (ClassificationDataSet method), 20

`splitWithProportion()` (SequentialDataSet method), 21
`splitWithProportion()` (SupervisedDataSet method), 22
`StateDependentLayer` (class in `pybrain.structure.modules`), 18
`SupervisedDataSet` (class in `pybrain.datasets.supervised`), 21
`SVMTrainer` (class in `pybrain.supervised.trainers.svmtrainer`), 22
`SVMUnit` (class in `pybrain.structure.modules.svmunit`), 22

T

`TanhLayer` (class in `pybrain.structure.modules`), 18
`testOnClassData()` (BackpropTrainer method), 26
`testOnSequenceData()` (in module `pybrain.tools.validation`), 26
`train()` (BackpropTrainer method), 26
`train()` (SVMTrainer method), 23
`trainEpochs()` (BackpropTrainer method), 27
`trainOnDataset()` (BackpropTrainer method), 27
`trainUntilConvergence()` (BackpropTrainer method), 27

V

`validate()` (CrossValidator method), 26
`validate` (ModuleValidator attribute), 25
`Validator` (class in `pybrain.tools.validation`), 25

W

`whichBuffers()` (FullConnection method), 17