PyBrain Documentation

Release 0.3

IDSIA/CogBotLab

November 18, 2009

CONTENTS

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 chapters can mostly be read independently 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 you want to develop for PyBrain and contribute, check out our guidelines in our wiki: http://wiki.github.com/pybrain/pybrain/guidelines.

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.

CHAPTER

ONE

INSTALLATION

Quick answer:

```
$ git clone git://github.com/pybrain/pybrain.git
$ python setup.py install
```

Long answer: We keep more detailed installation instructions (including dependencies) up-to-date in a wiki at http://wiki.github.com/pybrain/pybrain/installation.

CHAPTER

TWO

QUICKSTART

2.1 Building a Network

To go through the quickstart interactively, 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.

2.1.1 The buildNetwork Shortcut

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

```
>>>
```

2.1.3 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.

2.1.4 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.

2.2 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.

2.2.1 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.

2.2.2 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,))
```

2.2.3 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)
```

2.3 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.

CHAPTER

THREE

TUTORIALS

3.1 Introduction

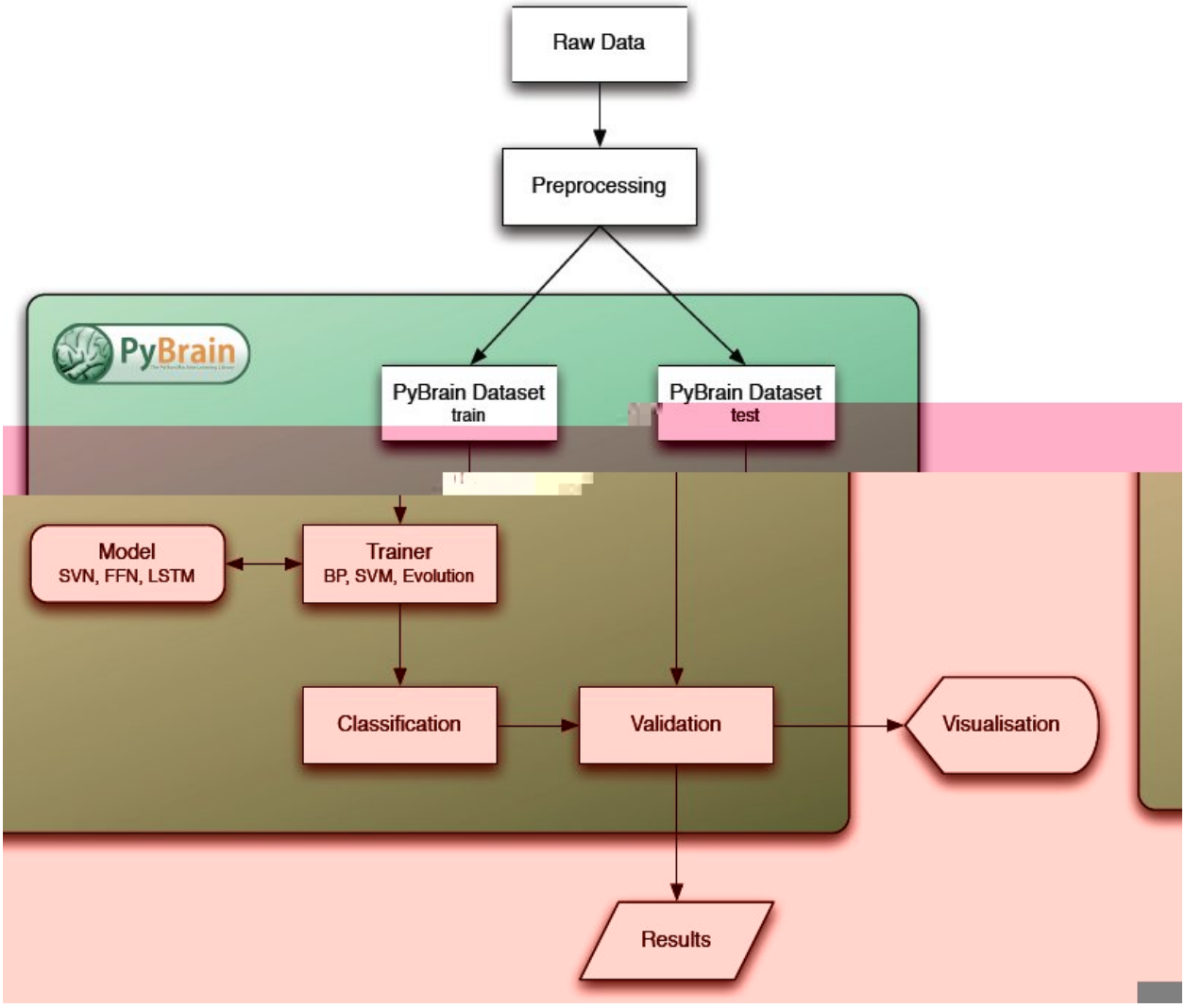
PyBrain's concept is to encapsulate different data processing algorithms in what we call a `Module`. A minimal Module contains a forward implementation depending on a collection of free parameters that can be adjusted, usually through some machine learning algorithm.

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.



3.2 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.

3.2.1 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.

3.2.2 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.

3.2.3 Naming your Networks structure

In some settings it makes sense to give the parts of a network explicit identifiers. The structural components are derive 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.

3.2.4 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.

3.3 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()
```

3.4 Using Datasets

Datasets are useful for allowing comfortable access to training, test and validation data. Instead of having to mangle with arrays, PyBrain gives you a more sophisticated datastructure that allows easier work with your data.

```
inp[0,:]
```

would yield the first input vector. In most cases there is also a field named 'target', which follows the same rules. However, didn't we say we will spare you the array mangling? Well, in most cases you will want iterate over a dataset like so:

```
for inp, targ in DS:
  ...
```

Note that whether you get one, two, or more sample rows as a return depends on the number of *linked fields* in the DataSet: These are fields containing the same number of samples and assumed to be used together, like the above 'input' and 'target' fields. You can always check the DS.link property to see which fields are linked.

Similarly, DataSets can be created by adding samples one-by-one – the cleaner but slower method – or by assembling them from arrays.

```
for inp, targ in samples:
    DS.appendLinked(inp, targ)
# or alternatively, with  ia  and  ta  being arrays:
assert(ia.shape[0] == ta.shape[0])
DS.setField('input', ia)
DS.setField('target', ta)
```

In the latter case DS cannot check the linked array dimensions for you, otherwise it would not be possible to build a dataset from scratch.

You may add your own linked or unlinked data to the dataset. However, note that many training algorithms iterate over the linked fields and may fail if their number has changed:

```
DS.addField('myfield')
DS.setField('myfield', myarray)
DS.linkFields('input','target','myfield') # must provide complete list here
```

A useful utility method for quick generation of randomly picked training and testing data is also provided:

```
>>> len(DS)
100
>>> TrainDS, TestDS = DS.splitWithProportion(0.8)
>>> len(TrainDS), len(TestDS)
(80, 20)
```

3.4.1 *supervised – Dataset for Supervised Regression Training*

As the name says, this simplest form of a dataset is meant to be used with supervised learning tasks. It is comprised of the fields 'input' and 'target', the pattern size of which must be set upon creation:

```
>>> from pybrain.datasets import SupervisedDataSet
>>> DS = SupervisedDataSet( 3, 2 )
>>> DS.appendLinked( [1,2,3], [4,5] )
>>> len(DS)
1
>>> DS['input']
array([[ 1.,  2.,  3.]])
```

3.4.2 *sequential – Dataset for Supervised Sequences Regression Training*

This dataset introduces the concept of `sequences`. With this we are moving further away from the array mangling towards something more practical for sequence learning tasks. Essentially, its patterns are subdivided into sequences of variable length, that can be accessed via the methods

```
getNumSequences()
getSequence(index)
getSequenceLength(index)
```

Creating a `Sequentialdataset` is no different from its parent, since it still contains only 'input' and 'target' fields. `Sequentialdataset` inherits from `SupervisedDataSet`, which can be seen as a special case with a sequence length of 1 for all sequences.

To fill the dataset with content, it is advisable to call `newSequence()` at the start of each sequence to be stored, and then add patterns by using `appendLinked()` as above. This way, the class handles indexing and such transparently. One can theoretically construct a `Sequentialdataset` directly from arrays, but messing with the index field is not recommended.

A typical way of iterating over a sequence dataset DS would be something like:

```
for i in range(DS.getNumSequences):
    for input, target in DS.getSequenceIterator(i):
       # do stuff
```

3.4.3 *classification – Datasets for Supervised Classification Training*

The purpose of this dataset is to facilitate dealing with classification problems, whereas the above are more geared towards regression. Its 'target' field is defined as integer, and it contains an extra field called 'class' which is basically an automated backup of the targets, for reasons that we be apparent shortly. For the most part, you don't have to bother with it. Initialization requires something like:

```
DS = ClassificationDataSet(inputdim, nb_classes=2, class_labels=['Fish','Chips'])
```

The labels are optional, and mainly used for documentation. Target dimension is supposed to be 1. The targets are class labels starting from zero. If for some reason you don't know beforehand how many you have, or you fiddled around with the `setField()` method, it is possible to regenerate the class information using `assignClasses()`, or `calculateStatistics()`:

```
>>> DS = ClassificationDataSet(2, class_labels=['Urd', 'Verdandi', 'Skuld'])
>>> DS.appendLinked([ 0.1, 0.5 ]   , [0])
>>> DS.appendLinked([ 1.2, 1.2 ]   , [1])
>>> DS.appendLinked([ 1.4, 1.6 ]   , [1])
>>> DS.appendLinked([ 1.6, 1.8 ]   , [1])
>>> DS.appendLinked([ 0.10, 0.80 ] , [2])
>>> DS.appendLinked([ 0.20, 0.90 ] , [2])

>>> DS.calculateStatistics()
{0: 1, 1: 3, 2: 2}
>>> print DS.classHist
{0: 1, 1: 3, 2: 2}
>>> print DS.nClasses
3
>>> print DS.getClass(1)
Verdandi
```

```
>>> print DS.getField('target').transpose()
[[0 1 1 1 2 2]]
```

When doing classification, many algorithms work better if classes are encoded into one output unit per class, that takes on a certain value if the class is present. As an advanced feature, `ClassificationDataSet` does this conversion automatically:

```
>>> DS._convertToOneOfMany(bounds=[0, 1])
>>> print DS.getField('target')
[[1 0 0]
 [0 1 0]
 [0 1 0]
 [0 1 0]
 [0 0 1]
 [0 0 1]]
>>> print DS.getField('class').transpose()
[[0 1 1 1 2 2]]
>>> DS._convertToClassNb()
>>> print DS.getField('target').transpose()
[[0 1 1 1 2 2]]
```

In case you want to do sequence classification, there is also a `SequenceClassificationDataSet`, which combines the features of this class and the `Sequentialdataset`.

3.4.4 *importance – Datasets for Weighted Supervised Training*

This is another extension of `Sequentialdataset` that allows assigning different weights to patterns. Essentially, it works like its parent, except comprising another linked field named 'importance', which should contain a value between 0.0 and 1.0 for each pattern. A `Sequentialdataset` is a special case with all weights equal to 1.0.

We have packed this functionality into a different class because it is rarely used and drains some computational resources. So far, there is no corresponding non-sequential dataset class.

3.5 Black-box Optimization

This tutorial will illustrate how to use the optimization algorithms in PyBrain.

Very many practical problems can be framed as optimization problems: finding the best settings for a controller, minimizing the risk of an investment portfolio, finding a good strategy in a game, etc. It always involves determining a certain number of *variables* (the *problem dimension*), each of them chosen from a set, that maximizing (or minimize) a given *objective function*.

The main categories of optimization problems are based on the kinds of sets the variables are chosen from:

- all real numbers: continuous optimization
- real numbers with bounds: constrained optimization
- integers: integer programming
- combinations of the above
- others, e.g. graphs

These can be further classified according to properties of the objective function (e.g. continuity, explicit access to partial derivatives, quadratic form, etc.). In black-box optimization the objective function is a black box, i.e. there are

no conditions about it. The optimization tools that PyBrain provides are all for the most general, black-box case. They fall into 2 groups:

- `BlackBoxOptimizer` are applicable to all kinds of variable sets
- `ContinuousOptimizer` can only be used for continuous optimization

We will introduce the optimization framework for the more restrictive kind first, because that case is simpler.

3.5.1 Continuous optimization

Let's start by defining a simple objective function for (`numpy` arrays of) continuous variables, e.g. the sum of squares:

```
>>> def objF(x): return sum(x**2)
```

and an initial guess for where to start looking:

```
>>> x0 = array([2.1, -1])
```

Now we can initialize one of the optimization algorithms, e.g. CMAES:

```
>>> from pybrain.optimization import CMAES
>>> l = CMAES(objF, x0)
```

By default, all optimization algorithms *maximize* the objective function, but you can change this by setting the `minimize` attribute:

```
>>> l.minimize = True
```

Note: We could also have done that upon construction: `CMAES(objF, x0, minimize = True)`

Stopping criteria can be algorithm-specific, but in addition, it is always possible to define the following ones:

- maximal number of evaluations
- maximal number of learning steps
- reaching a desired value

```
>>> l.maxEvaluations = 200
```

Now that the optimizer is set up, all we need to use is the `learn()` method, which will attempt to optimize the variables until a stopping criterion is reached. It returns a tuple with the best evaluable (= array of variables) found, and the corresponding fitness:

```
>>> l.learn()
(array([ -1.59778097e-05,  -1.14434779e-03]), 1.3097871509722648e-06)
```

3.5.2 General optimization: using `Evolvable`

Our approach to doing optimization in the most general setting (no assumptions about the variables) is to let the user define a subclass of `Evolvable` that implements:

- a `copy()` operator,
- a method for generating random other points: `randomize()`,
- `mutate()`, an operator that does a small step in search space, according to *some* distance metric,

- (optionally) a `crossover()` operator that produces *some* combination with other evolvables of the same class.

The optimization algorithm is then initialized with an instance of this class and an objective function that can evaluate such instances.

Here's a minimalistic example of such a subclass with a single constrained variable (and a bias to do mutation steps toward larger values):

```
>>> from random import random
>>> from pybrain.structure.evolvables.evolvable import Evolvable
>>> class SimpleEvo(Evolvable):
...     def __init__(self, x): self.x = max(0, min(x, 10))
...     def mutate(self):      self.x = max(0, min(self.x + random() - 0.3, 10))
...     def copy(self):        return SimpleEvo(self.x)
...     def randomize(self):   self.x = 10*random()
...     def __repr__(self):    return '<-%.2f->'+str(self.x)
```

which can be optimized using, for example, `HillClimber`:

```
>>> from pybrain.optimization import HillClimber
>>> x0 = SimpleEvo(1.2)
>>> l = HillClimber(lambda x: x.x, x0, maxEvaluations = 50)
>>> l.learn()
(<-10.00->, 10)
```

3.5.3 Optimization in Reinforcement Learning

This section illustrates how to use optimization algorithms in the reinforcement learning framework.

As our objective function we use any episodic task, e.g:

```
>>> from pybrain.rl.environments.cartpole.balancetask import BalanceTask
>>> task = BalanceTask()
```

Then we construct a module that can interact with the task, for example a neural network controller,

```
>>> from pybrain.tools.shortcuts import buildNetwork
>>> net = buildNetwork(task.outdim, 3, task.indim)
```

and we choose any optimization algorithm, e.g. a simple `HillClimber`.

Now, we have 2 (equivalent) ways for connecting those:

1. **using the same syntax as before, where the task plays the role of the objective function directly:**

   ```
   >>> HillClimber(task, net, maxEvaluations = 100).learn()
   ```

2. **or, using the agent-based framework:**

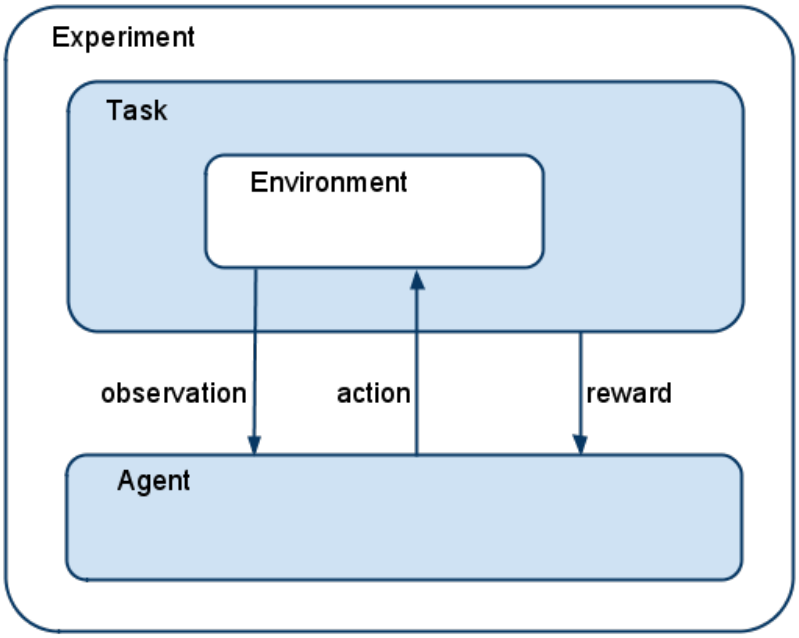
   ```
   >>> from pybrain.rl.agents import OptimizationAgent
   >>> from pybrain.rl.experiments import EpisodicExperiment
   >>> agent = OptimizationAgent(net, HillClimber())
   >>> exp = EpisodicExperiment(task, agent)
   >>> exp.doEpisodes(100)
   ```

Note: This is very similar to the typical (non-optimization) reinforcement learning setup, the key difference being the use of a `LearningAgent` instead of an `OptimizationAgent`.

```
>>> from pybrain.rl.learners import ENAC
>>> from pybrain.rl.agents import LearningAgent
>>> agent = LearningAgent(net, ENAC())
>>> exp = EpisodicExperiment(task, agent)
>>> exp.doEpisodes(100)
```

3.6 Reinforcement Learning

A reinforcement learning (RL) task in PyBrain always consists of a few components that interact with each other: `Environment`, `Agent`, `Task`, and `Experiment`. In this tutorial we will go through each of them, create the instances and explain what they do.



But first of all, we need to import some general packages and the RL components from PyBrain:

```
from scipy import *
import sys, time

from pybrain.rl.environments.mazes import Maze, MDPMazeTask
from pybrain.rl.learners.valuebased import ActionValueTable
from pybrain.rl.agents import LearningAgent
from pybrain.rl.learners import Q, SARSA
```

```
from pybrain.rl.experiments import Experiment
from pybrain.rl.environments import Task
```

Note: You can directly run the code in this tutorial by running the script `docs/tutorials/rl.py`.

For later visualization purposes, we also need to initialize the plotting engine (interactive mode).

```
import pylab
pylab.gray()
pylab.ion()
```

The `Environment` is the world, in which the agent acts. It receives input with the `performAction()` method and returns an output with `getSensors()`. All environments in PyBrain are located under `pybrain/rl/environments`.

One of these environments is the maze environment, which we will use for this tutorial. It creates a labyrinth with free fields, walls, and an goal point. An agent can move over the free fields and needs to find the goal point. Let's define the maze structure, a simple 2D numpy array, where 1 is a wall and 0 is a free field:

```
structure = array([[1, 1, 1, 1, 1, 1, 1, 1, 1],
                   [1, 0, 0, 1, 0, 0, 0, 0, 1],
                   [1, 0, 0, 1, 0, 0, 1, 0, 1],
                   [1, 0, 0, 1, 0, 0, 1, 0, 1],
                   [1, 0, 0, 1, 0, 1, 1, 0, 1],
                   [1, 0, 0, 0, 0, 0, 1, 0, 1],
                   [1, 1, 1, 1, 1, 1, 1, 0, 1],
                   [1, 0, 0, 0, 0, 0, 0, 0, 1],
                   [1, 1, 1, 1, 1, 1, 1, 1, 1]])
```

Then we create the environment with the structure as first parameter and the goal field tuple as second parameter:

```
environment = Maze(structure, (7, 7))
```

Next, we need an agent. The agent is where the learning happens. It can interact with the environment with its `getAction()` and `integrateObservation()` methods.

The agent itself consists of a controller, which maps states to actions, a learner, which updates the controller parameters according to the interaction it had with the world, and an explorer, which adds some explorative behavior to the actions. All standard agents already have a default explorer, so we don't need to take care of that in this tutorial.

The controller in PyBrain is a module, that takes states as inputs and transforms them into actions. For value-based methods, like the Q-Learning algorithm we will use here, we need a module that implements the `ActionValueInterface`. There are currently two modules in PyBrain that do this: The `ActionValueTable` for discrete actions and the `ActionValueNetwork` for continuous actions. Our maze uses discrete actions, so we need a table:

```
controller = ActionValueTable(81, 4)
controller.initialize(1.)
```

The table needs the number of states and actions as parameters. The standard maze environment comes with the following 4 actions: north, south, east, west.

Then, we initialize the table with 1 everywhere. This is not always necessary but will help converge faster, because unvisited state-action pairs have a promising positive value and will be preferred over visited ones that didn't lead to the goal.

Each agent also has a learner component. Several classes of RL learners are currently implemented in PyBrain: black box optimizers, direct search methods, and value-based learners. The classical Reinforcement Learning mostly

consists of value-based learning, in which of the most well-known algorithms is the Q-Learning algorithm. Let's now create the agent and give it the controller and learner as parameters.

```
learner =
```

3.7.1 Layers

We will now show how to implement new types of layers, which are a very simple form of a module. They can be used to implement transfer functions, resulting in special layers such as the `SigmoidLayer`.

The `NeuronLayer` class serves as a base class for all types of layers. Let's have a look at how a very basic layer, the `LinearLayer` looks like and implement a specific new type afterwards:

```
from neuronlayer import NeuronLayer


class LinearLayer(NeuronLayer):
    """ The simplest kind of module, not doing any transformation. """

    def _forwardImplementation(self, inbuf, outbuf):
        outbuf[:] = inbuf

    def _backwardImplementation(self, outerr, inerr, outbuf, inbuf):
        inerr[:] = outerr
```

As we can see, Layer class relies on two methods: `_forwardImplementation()` and `_backwardImplementation()`. (Note the leading underscores which are a Python convention to indicate pseudo-private methods.)

The first method takes two arguments, `inbuf` and `outbuf`. Both are Scipy arrays of the size of the layer's input and output dimension respectively. The arrays have already been created for us. The pybrain structure framework now expects us to produce an output from the input and put it into `outbuf` in place.

Note: Manipulating an array in place with SciPy works via the `[:]` operator. Given an array a, the line `a[:] = b` will overwrite a's memory with the contents of b.

The second method is used to calculate the derivative of the output error with respect to the input. From standard texts on neural networks, we know that the error of a unit (which is a field in a layer in our case) can be calculated as the derivative of the unit's transfer function's applied to the units input multiplied with the error. In the case of the identity, the derivative is a constant 1 and thus backpropagating the error is nothing but just copying it.

Thus, any `_backwardImplementation()` implementation must fill `inerror` correctly.

An example: quadratic polynomial as transfer function

For the sake of simplicity, let's implement a layer that uses $f(x) = x^2$ as a transfer function. The derivative is then given by $f'(x) = 2x$.

Let's start out with the pure skeleton of the class:

```
from pybrain.structure.modules.neuronlayer import NeuronLayer

class QuadraticPolynomialLayer(NeuronLayer):

    def _forwardImplementation(self, inbuf, outbuf):
      pass

    def _backwardImplementation(self, outerr, inerr, outbuf, inbuf):
      pass
```

The current methods don't do anything, so let's implement one after the other.

Using SciPy, we can use most of Python's arithmetic syntax directly on the array and it is applied component wise to it. Thus, to get the square of an array `a` we can just call `a**2`. Thus:

```
def _forwardImplementation(self, inbuf, outbuf):
  outbuf[:] = inbuf**2
```

Remembering Neural Networks 101, we can implement the derivative as such:

```
def _backwardImplementation(self, outerr, inerr, outbuf, inbuf):
  inerr[:] = 2 * inbuf * outerr
```

Further readings: Layers can be used in numerous different ways. The interested reader is referred to the source code of the `LSTMLayer` and the mailing list for further help.

3.7.2 Connections

A `Connection` works similarly to a Layer in many ways. The key difference is that a Connection processes data it does not "own", meaning that their primary task is to shift data from one node of the network graph to another.

The API is similar to that of Layers, having `_forwardImplementation()` and `_backwardImplementation()` methods. However, sanity checks are performed in the constructors mostly to assure that the modules connected can actually be connected.

As an example, we will now have a look at the `IdentityConnection` and afterwards implement a trivial example.

IdentityConnections are just pushing the output of one module into the input of another. They do not transform it in any way.

```
1  from connection import Connection
2
3  class IdentityConnection(Connection):
4      """Connection which connects the i'th element from the first module's output
5      buffer to the i'th element of the second module's input buffer."""
6
7      def __init__(self, *args, **kwargs):
8          Connection.__init__(self, *args, **kwargs)
9          assert self.indim == self.outdim, \
10                 "Indim (%i) does not equal outdim (%i)" % (
11                 self.indim, self.outdim)
12
13     def _forwardImplementation(self, inbuf, outbuf):
14         outbuf += inbuf
15
16     def _backwardImplementation(self, outerr, inerr, inbuf):
17         inerr += outerr
```

The constructor basically does two things. First, it calls the constructor of the superclass in order to perform any bookkeeping. After that, it asserts that the incoming and the outgoing module (the modules that are connected by the connection) are actually compatible, which means that the input dimension of the outgoing module equals the output dimension of the incoming module.

The `_forwardImplementation()` is called with the output buffer of the incoming module, depicted as `inbuf`, and the input buffer of the outgoing module, called `outbuf`. You can think of the two as a source and a sink. Mind that in line #14, we actually do not overwrite the buffer but instead perform an addition. This is because there might be other modules connected to the outgoing module. The buffers will be overwritten with by the `Network` instance containing the modules and connections.

The `_backwardImplementation()` works similarly. However, connections do also get the `inbuf` buffer since in some cases (like `FullConnection` instances) it is necessary to calculate derivatives in order to adapt parameters.

3.7.3 ParameterContainers

In all neural networks, you want adaptable parameters that can be trained by an external learning algorithm. Structure that holds those parameters are realized via subclassing `ParameterContainer`. Due to Python's ability of multiple inheritance, you can just add this to the list of your subclasses. We will now look at this with the `FullConnection` as an example. The linear connection is a connection which connects two layers and processes the data by multiplication with a weight matrix. This kind of connection is the most common in neural networks.

```
1  from scipy import reshape, dot, outer
2
3  from connection import Connection
4  from pybrain.structure.parametercontainer import ParameterContainer
5
6
7  class FullConnection(Connection, ParameterContainer):
8
9      def __init__(self, *args, **kwargs):
10         Connection.__init__(self, *args, **kwargs)
11         ParameterContainer.__init__(self, self.indim*self.outdim)
12
13     def _forwardImplementation(self, inbuf, outbuf):
14         outbuf += dot(reshape(self.params, (self.outdim, self.indim)), inbuf)
15
16     def _backwardImplementation(self, outerr, inerr, inbuf):
17         inerr += dot(reshape(self.params, (self.outdim, self.indim)).T, outerr)
18         self.derivs += outer(inbuf, outerr).T.flatten()
```

In line 10 and 11 we can see how the superclasses' constructors are called. `ParameterContainer` expects an integer argument N, which depicts the amount of parameters the FullConnection needs, which is the product of the incoming modules size and the outgoing modules size.

Due this, the constructor of ParameterContainer gives the object two fields: `params` and `derivs` which are two arrays of size N. These are used to hold parameters and possibly derivatives.

In the case of backpropagation, learning happens during calls to `_backwardImplementation()`. In line 18, we can see how the field `derivs` is modified.

3.7.4 Checking for correctness

Remembering Neural Networks 102, we know that we can check the gradients of our neural network implementation numerically. PyBrain already comes with a function that does exactly that, `gradientCheck()`. You can pass it any network containing a structural component you programed and it will check whether the numerical gradients are (roughly) equal to the gradient given by the `_backwardImplementation()` methods.

So let's check our new QuadraticPolynomialLayer.

```
1  from pybrain.tools.shortcuts import buildNetwork
2  from pybrain.tests.helpers import gradientCheck
3
4  n = buildNetwork(2, 3, 1, hiddenclass=QuadraticPolynomialLayer)
5  n.randomize()
6  gradientCheck(n)
```

First we do the necessary imports in line 1 and 2. Then we build a network with our special class in line 4. To initialize the weights of the network, we randomize its parameters in line 5 and call our gradient check in line 6. If we have done everything right, we will be rewarded with the output `Perfect gradient`.

CHAPTER

FOUR

ADVANCED

4.1 Fast networks for PyBrain

Writing a neural networking framework in Python imposes certain speed restrictions upon it. Python is just not as fast as languages such as C++ or Fortran.

Due to this, PyBrain has its own spin-off called arac, which is a re-implementation of its neural networking facilities that integrates transparently with it.

Depending on the configuration of your system, speedups of **5-10x faster** can be expected. This speedup might be even higher (ranging into the hundreds) if you use sophisticated topologies such as MDRNNs. If you want some numbers on your system, there is a comparison script shipped with PyBrain at `examples/arac/benchmark.py`.

4.1.1 Installation

However, the installation process is less easy than for pure Python PyBrain. On the other hand, it's mostly about installing an additional library and telling PyBrain to use fast networks.

You will find detailed installation instructions for arac on the arac wiki.

This instructions are work in progress. If you run into difficulties, ask on the PyBrain mailing list.

4.1.2 Usage

Once you have installed it, there are three ways to use fast networks.

First, the shortcut `buildNetwork` has a keyword `fast` which builds an arac network instead:

```
>>> from pybrain.tools.shortcuts import buildNetwork
>>> n = buildNetwork(2, 3, 1, fast=True)
>>> n.activate((2, 3))
array([-0.20781205])
```

As you can see by examining the network, it is a special class:

```
>>> n
<_FeedForwardNetwork '_FeedForwardNetwork-8'>
```

which is prefixed with an underscore, the Python convention for naming C implementations of already existing classes. We can import these classes directly from arac and use them in the same way as PyBrain classes:

```
>>> from arac.pybrainbridge import _FeedForwardNetwork, _RecurrentNetwork
```

The third method is to construct a network as a PyBrain network and call the method `convertToFastNetwork` afterwards:

```
>>> n = buildNetwork(2, 3, 1, fast=False)
>>> n.convertToFastNetwork()
<_FeedForwardNetwork '_FeedForwardNetwork-18'>
```

However, be cautious with the last method since changes to the PyBrain network are not reflected in the arac network.

4.1.3 Limitations

Since arac is implemented in C++ and currently maintained by only a single person, arac development is likely to be slower than PyBrain's. This might lead to certain features (e.g. layer types) to be implemented later than the pure python versions. This also applies to custom layer types. As soon as you have your layer type, you will not be able to use fast networks anymore – except if you chose to also implement them for arac yourself.

4.2 Using ODE Environments

4.2.1 Using an existing ODE environment

This tutorial walks you through the process of setting up an existing ODE Environment for use as a testbed for RL or optimization algorithms.

First we need the following additional packages that are not required for PyBrain (in addition to SciPy):

- matplotlib
- python-tk
- python-pyode
- python-opengl (if you also want to view what is happening, very recommended)

You also need to exchange the following two `.py` files with custom versions:

```
cd pybrain/pybrain/rl/environments/ode/xode_changes/
sudo cp * /usr/lib/python2.6/dist-packages/xode/ (or there ever your dist-packages are)
```

You can test if all your settings are ok by starting following example:

```
cd ~/pybrain/examples/rl/
python johnnie_pgpe.py
```

... and then view what is happening by using the viewer:

```
cd ~/pybrain/pybrain/rl/environments/ode
python viewer.py
```

Note: On Linux, if that gives rise to a segmentation fault, try installing `xorg-driver-fglrx`

Existing ODE Environments that are tested are:

- Johnnie (a biped humanoid robot modeled after the real robot Johnnie (http://www.amm.mw.tum.de)

- CCRL (a robot with two 7 DoF Arms and simple grippers, modeled after the real robot at the CCRL of TU Munich. (http://www.lsr.ei.tum.de/)
- PencilBalancer (a robot that balances pencils in a 2D way, modeled after the real robot from Jörg Conradt. (http://www.ini.uzh.ch/~conradt/Projects/PencilBalancer/)

4.2.2 Creating your own learning task in an existing ODE environment

This tutorial walks you through the process of setting up a new task within an existing ODE Environment. It assumes that you have taken the steps described in the section *Using an existing ODE environment*.

For all ODE environments there can be found a standard task in `pybrain/rl/environments/ode/tasks`

We take as an example again the Johnnie environment. You will find that the first class in the johnnie.py file in the above described location is named JohnnieTask and inherits from EpisodicTask.

The necessary methods that you need to define your own task are described already in that basic class:

- `__init__(self, env)` - the constructor
- `performAction(self, action)` - processes and filters the output from the controller and communicates it to the environment.
- `isFinished(self)` - checks if the maximum number of timesteps has been reached or if other break condition has been met.
- `res(self)` - resets counters rewards and similar.

If we take a look at the StandingTask (the next class in the file) we see that only little has to be done to create an own task. First of all the class must inherit from JohnnieTask. Then, the constructor has to be overwritten to declare some variables and constants for the specific task. In this case there were some additional position sensors added and normalized for reward calculation. As normally last step the getReward Method has to be overwritten, because the reward definition is normally what defines the task. In this case just the vertical head position is returned (with some clipping to prevent the robot from jumping to get more reward). That is already enough to create a task that is sufficiently defined to make a proper learning method (like PGPE in the above mentioned and testable example johnnie_pgpe.py) learn a controller that let the robot stand complete upright without falling.

For some special cases you maybe are forced to rewrite the performAction method and the isFinished method, but that special cases are out of scope of this HowTo. If you need to make such changes and encounter problems please feel free to contact the PyBrain mailing list.

4.2.3 Creating your own ODE environment

This tutorial walks you through the process of setting up a new ODE Environment. It assumes that you are already familiar with the sections *Using an existing ODE environment* and *Creating your own learning task in an existing ODE environment* and have taken the necessary steps explained there.

If you want to your own environment you need the following:

- Environment that inherits from ODEEnvironment
- Agent that inherits from OptimizationAgent
- Tasks that inherit from EpisodicTask

For all ODE environments, an instance can be found in `pybrain/rl/environments/ode/instances/`

We take as an example again the Johnnie environment. You will find that the first class in the `johnnie.py` file in the location described above is named `JohnnieEnvironment` and inherits from `ODEEnvironment`.

You will see that were is not much to do on the PyBrain side to generate the environment class. First loading the corresponding XODE file is necessary to provide PyBrain with the specification of the simulation. How to generate the corresponding XODE file will be shown later in this HowTo. Then the standard sensors are added like the JointSensors, the corresponding JointVelocitySensors and also the actuators for every joint. Because this kind of sensors and actuators are needed in every simulation they are already added in the environment and assumed to exist by later stages of PyBrain.

The next part is a bit more involved. First, member variables that state the number of action dimensions and number of sensors have to be set.

```
self.actLen = self.getActionLength()
self.obsLen = len(self.getSensors())
```

Next, 3 lists are generated for every action dimension. The first list is called `torqueList` and states the fraction of the global maximal force that can bee applied to the joints. The second list states the maximum angle, the third list states the minimum angle for every joint. (`cHighList` and `cLowList`) For example:

```
self.tourqueList = array([0.2, 0.2, 0.2, 0.5, 0.5, 2.0, 2.0,2.0,2.0,0.5,0.5],)
self.cHighList = array([1.0, 1.0, 0.5, 0.5, 0.5, 1.5, 1.5,1.5,1.5,0.25,0.25],)
self.cLowList = array([-0.5, -0.5, -0.5, 0.0, 0.0, 0.0, 0.0,0.0,0.0,-0.25,-0.25],)
```

The last thing to do is how much simulation steps ODE should make before getting an update from the controller and sending new sensor values back, called stepsPerAction.

4.2.4 Creating your own XODE instance

Now we want to specify a instantiation in a XODE file. If you do not know ODE very well, you can use a script that is shipped with PyBrain and can be found in `pybrain/rl/environments/ode/tools/xodetools.py`

The first part of the file is responsible for parsing the simplified XODE code to a regular XODE file, that can be ignored. For an example, look at the Johnnie definition by searching for `class XODEJohnnie(XODEfile)`

The instantiation of what you want to simulate in ODE is defined in this tool as a class that inherits from `XODEfile`. The class consists only of a constructor. Here all parts of the simulated object are defined. The parts are defined in an global coordinate system. For examples the row

```
self.insertBody('arm_left','cappedCylinder',[0.25,7.5],5,pos=[2.06,-2.89,0],
                                euler=[90,0,0], passSet=['total'], mass=2.473)
```

creates the left arm (identifier 'arm_left') of Johnnie as an cylinder with round endings ('cappedCylinder') with a diameter of 0.25 and a length of 7.5 ([0.25,7.5]) with a density of 5 (that will be overwritten if the optional value mass is given at the end of the command), an initial position of `pos = [2.06,-2.89,0]`, turned by 90 degrees around the x-Axis (`euler = [90,0,0]`, all capped cylinders are by default aligned with the y-Axis) the passSet named 'total' (will be explained soon) and the optional mass of the part.

"passSet" is used to define parts that can penetrate each other. That is especially necessary for parts that have a joint together, but can also be usable in other cases. All parts that are part of the same passSet can penetrate each other. Multiple passSet names can be given delimited by a ",". Types that are understood by this tool are:

- cylinder
- cappedCylinder
- box

Next we have to define the joints that connect the parts. Types of joints that are understood by this tool are:

- fixed, for a stiff fixed joint.

- hinge, one dimensional joint.
- universal joint, experimental 2D joint.

A joint between two parts is inserted in the model by insertJoint, giving the identifier of the first part, then the identifier of the second part. Next the type of joint is stated (e.g. 'hinge'). The axis around the joint will rotate is stated like `axis={'x':1,'y':0,'z':0}` and the anchor point in global coordinates is defined by something like `anchor=(2.06,0.86,0)`. Add all parts and joints for your model.

Finally with `centerOn(identifier)` the camera position is fixed to that part and with `insertFloor(y=??)` a floor can be added.

Now go to the end of the file and state:

```
name = YourClass('../models/name')
name.writeXODE()
```

and execute the file with

```
python xodetools.py
```

And you have created an instantiation of your model that can be read in in the above environment.

What is missing is a default task for the new environment. In the previous "HowTo create your own learning task in an existing ODE environment" we saw how such a standard task looks for the Johnnie environment. To create our own task we have to create a file with the name of our environment in `pybrain/rl/environments/ode/tasks/`

The new task has to import the following packages:

from pybrain.rl.environments import EpisodicTask from pybrain.rl.environments.ode.sensors import *

And whatever is needed from scipy and similar.

The new class should inherit from EpisodicTask like in the JohnnieTask. Next we create the constructor that takes the environment with `def __init__(self, env)`.

It is important that the constructor of EpisodicTask is called.

```
EpisodicTask.__init__(self, env)
```

The following member variables are mandatory:

```
self.maxPower = 100.0   #Overall maximal torque - is multiplied with relative max
                        #torque for individual joint to get individual max torque
self.reward_history = []
self.count = 0          #timestep counter
self.epiLen = 500       #time steps for one episode
```

In contrast to the ODEEnvironment standard settings some changes might be needed:

- `self.env.FricMu` if you need higher or lower friction for your task,
- `self.env.dt` if you need more timely resolution.

Next the sensor and actuator limits must be set, usually between -1 and 1:

```
# normalize standard sensors to (-1, 1)
self.sensor_limits = []
#Angle sensors
for i in range(self.env.actLen):
    # Joint velocity sensors
```

```
    self.sensor_limits.append((self.env.cLowList[i], self.env.cHighList[i]))
for i in range(self.env.actLen):
    self.sensor_limits.append((-20, 20))
#Normalize all actor dimensions to (-1, 1)
self.actor_limits = [(-1, 1)]*env.actLen
```

The next method that is needed is the performAction method, the standard setting looks like that:

```
def performAction(self, action):
    """ Filtered mapping towards performAction of the underlying environment """
    EpisodicTask.performAction(self, action)
```

If you want to control the wanted angels instead of the forces you may include this simple PD mechanism:

```
#The joint angles
isJoints = self.env.getSensorByName('JointSensor')
#The joint angular velocities
isSpeeds = self.env.getSensorByName('JointVelocitySensor')
#norm output to action interval
act = (action+1.0)/2.0*(self.env.cHighList-self.env.cLowList)+self.env.cLowList
#simple PID
action = tanh((act - isJoints - isSpeeds) * 16.0) * self.maxPower * self.env.tourqueList
```

Now we have to define the `isFinished()` method:

```
def isFinished(self):
    """ returns true if episode timesteps has reached episode length and resets the task """
    if self.count > self.epiLen:
        self.res()
        return True
    else:
        self.count += 1
        return False
```

You are certainly free to include other breaking conditions.

Finally we define a `reset()` method:

```
def res(self
```

CHAPTER

FIVE

API

5.1 `connections` – Structural Components: Connections

class **`Connection`**(*inmod, outmod, name=None, inSliceFrom=0, inSliceTo=None, outSliceFrom=0, outSliceTo=None*)

A connection links 2 modules, more precisely: the output of the first module to the input of the second. It can potentially transform the information on the way. It also transmits errors backwards between the same modules.

`__init__`(*inmod, outmod, name=None, inSliceFrom=0, inSliceTo=None, outSliceFrom=0, outSliceTo=None*)

Every connection requires an input and an output module. Optionally, it is possible to define slices on the buffers.

Parameters

- *inmod* – input module
- *outmod* – output module

Key insliceform starting index on the buffer of inmod (default = 0)

Key insliceto ending index on the buffer of inmod (default = last)

Key outslicefrom starting index on the buffer of outmod (default = 0)

Key outsliceto ending index on the buffer of outmod (default = last)

`forward`(*inmodOffset=0, outmodOffset=0*)

Propagate the information from the incoming module's output buffer, adding it to the outgoing node's input buffer, and possibly transforming it on the way.

For this transformation use inmodOffset as an offset for the inmod and outmodOffset as an offset for the outmodules offset.

`backward`(*inmodOffset=0, outmodOffset=0*)

Propagate the error found at the outgoing module, adding it to the incoming module's output-error buffer and doing the inverse transformation of forward propagation.

For this transformation use inmodOffset as an offset for the inmod and outmodOffset as an offset for the outmodules offset.

If appropriate, also compute the parameter derivatives.

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

Bases: `pybrain.structure.connections.connection.Connection`, `pybrain.structure.parametercontainer.ParameterContainer`

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*)
Bases: pybrain.structure.connections.connection.Connection

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 MotherConnection(*nbparams, **args*)
Bases: pybrain.structure.parametercontainer.ParameterContainer

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

class SharedConnection(*mother, *args, **kwargs*)
Bases: pybrain.structure.connections.connection.Connection

A shared connection can link different couples of modules, with a single set of parameters (encapsulated in a MotherConnection).

mother
pointer to MotherConnection

class SharedFullConnection(*mother, *args, **kwargs*)
Bases: pybrain.structure.connections.shared.SharedConnection, pybrain.structure.connections.full.FullConnection

Shared version of FullConnection.

5.2 evolvable – Container Component: Evolvable

class Evolvable()
The interface for all Evolvables, i.e. which implement mutation, randomize and copy operators.

copy()
By default, returns a full deep-copy - subclasses should implement something faster, if appropriate.

mutate(***args*)
Vary some properties of the underlying module, so that it's behavior changes, (but not too abruptly).

newSimilarInstance()
Generates a new Evolvable of the same kind.

randomize()
Sets all variable parameters to random values.

5.3 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

`__init__`(*dim, peepholes=False, name=None*)

Parameter *dim* – number of cells

Key peepholes enable peephole connections (from state to gates)?

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 statte

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.

5.4 networks – Structural Components: Networks

class **Network**(*name=None, **args*)
Bases: pybrain.structure.modules.module.Module, pybrain.structure.parametercontainer.Param

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.

5.5 actionvalues – RL Components: ActionValues

class ActionValueInterface()
Interface for different ActionValue modules, like the ActionValueTable or the ActionValueNetwork.

class ActionValueTable(*numStates, numActions, name=None*)
Bases: pybrain.structure.modules.table.Table, pybrain.rl.learners.valuebased.interface.Act

A special table that is used for Value Estimation methods in Reinforcement Learning.

getMaxAction(*state*)
Return the action with the maximal value for the given state.

initialize(*value=0.0*)
Initialize the whole table with the given value.

class ActionValueNetwork(*dimState, numActions, name=None*)
Bases: pybrain.structure.modules.module.Module, pybrain.rl.learners.valuebased.interface.A

getActionValues(*state*)
Run forward activation for each of the actions and returns all values.

getMaxAction(*state*)
Return the action with the maximal value for the given state.

5.6 agents – RL Components: Agents

class Agent()
An agent is an entity capable of producing actions, based on previous observations. Generally it will also learn from experience. It can interact directly with a Task.

getAction()
Return a chosen action. :rtype: by default, this is assumed to ba a numpy array of doubles. :note: This method is abstract and needs to be implemented.

giveReward(*r*)
Reward or punish the agent. :key r: reward, if C{r} is positive, punishment if C{r} is negative :type r: double

integrateObservation(*obs*)
Integrate the current observation of the environment. :arg obs: The last observation returned from the environment :type obs: by default, this is assumed to be a numpy array of doubles

newEpisode()
Inform the agent that a new episode has started.

class LoggingAgent(*indim, outdim*)
Bases: pybrain.rl.agents.agent.Agent

This agent stores actions, states, and rewards encountered during interaction with an environment in a ReinforcementDataSet (which is a variation of SequentialDataSet). The stored history can be used for learning and is erased by resetting the agent. It also makes sure that integrateObservation, getAction and giveReward are called in exactly that order.

getAction()
Step 2: store the action in a temporary variable until reward is given.

giveReward(*r*)
Step 3: store observation, action and reward in the history dataset.

integrateObservation(*obs*)
Step 1: store the observation received in a temporary variable until action is called and reward is given.

newEpisode()
Indicate the beginning of a new episode in the training cycle.

reset()
Clear the history of the agent.

class **LearningAgent**(*module, learner=None*)
Bases: pybrain.rl.agents.logging.LoggingAgent

LearningAgent has a module, a learner, that modifies the module, and an explorer, which perturbs the actions. It can have learning enabled or disabled and can be used continuously or with episodes.

getAction()
Activate the module with the last observation, adda the exploration from the explorer object and store the result as last action.

learn(*episodes=1*)
Call the learner's learn method, which has access to both module and history.

learning
Return whether the agent currently learns from experience or not.

newEpisode()
Indicate the beginning of a new episode in the training cycle.

reset()
Clear the history of the agent and resets the module and learner.

class **OptimizationAgent**(*module, learner*)
Bases: pybrain.rl.agents.agent.Agent

A simple wrapper to allow optimizers to conform to the RL interface. Works only in conjunction with EpisodicExperiment.

5.7 experiments – RL Components: Experiments

class **Experiment**(*task, agent*)
An experiment matches up a task with an agent and handles their interactions.

doInteractions(*number=1*)
The default implementation directly maps the methods of the agent and the task. Returns the number of interactions done.

class **EpisodicExperiment**(*task, agent*)
Bases: pybrain.rl.experiments.experiment.Experiment

The extension of Experiment to handle episodic tasks.

doEpisodes(*number=1*)
Do one episode, and return the rewards of each step as a list.

class **ContinuousExperiment**(*task, agent*)
Bases: pybrain.rl.experiments.experiment.Experiment

The extension of Experiment to handle continuous tasks.

doInteractionsAndLearn(*number=1*)
Execute a number of steps while learning continuously. no reset is performed, such that consecutive calls to this function can be made.

5.8 explorers – RL Components: Explorers

class **Explorer**(*indim, outdim, name=None, **args*)
An Explorer object is used in Agents, receives the current state and action (from the controller Module) and returns an explorative action that is executed instead the given action.

activate(*state, action*)
The super class commonly ignores the state and simply passes the action through the module. implement _forwardImplementation() in subclasses.

newEpisode()
Inform the explorer about the start of a new episode.

5.8.1 Continuous Explorers

class **NormalExplorer**(*dim, sigma=0.0*)
A continuous explorer, that perturbs the resulting action with additive, normally distributed random noise. The exploration has parameter(s) sigma, which are related to the distribution's standard deviation. In order to allow for negative values of sigma, the real std. derivation is a transformation of sigma according to the expln() function (see pybrain.tools.functions).

class **StateDependentExplorer**(*statedim, actiondim, sigma=-2.0*)
A continuous explorer, that perturbs the resulting action with additive, normally distributed random noise. The exploration has parameter(s) sigma, which are related to the distribution's standard deviation. In order to allow for negative values of sigma, the real std. derivation is a transformation of sigma according to the expln() function (see pybrain.tools.functions).

activate(*state, action*)
The super class commonly ignores the state and simply passes the action through the module. implement _forwardImplementation() in subclasses.

newEpisode()
Randomize the matrix values for exploration during one episode.

5.8.2 Discrete Explorers

class **DiscreteExplorer**()
Bases: pybrain.rl.explorers.explorer.Explorer

Discrete explorers choose one of the available actions from the set of actions. In order to know which actions are available and which action to choose, discrete explorers need access to the module (which has to of class ActionValueTable).

_setModule(*module*)
Tells the explorer the module (which has to be ActionValueTable).

class **EpsilonGreedyExplorer**(*epsilon=0.29999999999999999, decay=0.99990000000000001*)
Bases: pybrain.rl.explorers.discrete.discrete.DiscreteExplorer

A discrete explorer, that executes the original policy in most cases, but sometimes returns a random action (uniformly drawn) instead. The randomness is controlled by a parameter 0 <= epsilon <= 1. The closer epsilon gets to 0, the more greedy (and less explorative) the agent behaves.

_forwardImplementation(*inbuf, outbuf*)
Draws a random number between 0 and 1. If the number is less than epsilon, a random action is chosen. If it is equal or larger than epsilon, the greedy action is returned.

class **BoltzmannExplorer**(*tau=2.0, decay=0.99950000000000006*)
Bases: pybrain.rl.explorers.discrete.discrete.DiscreteExplorer

A discrete explorer, that executes the actions with probability that depends on their action values. The boltzmann explorer has a parameter tau (the temperature). for high tau, the actions are nearly equiprobable. for tau close to 0, this action selection becomes greedy.

activate(*state, action*)
The super class ignores the state and simply passes the action through the module. implement _forwardImplementation() in subclasses.

_forwardImplementation(*inbuf, outbuf*)
Draws a random number between 0 and 1. If the number is less than epsilon, a random action is chosen. If it is equal or larger than epsilon, the greedy action is returned.

class **DiscreteStateDependentExplorer**(*epsilon=0.20000000000000001, decay=0.99980000000000002*)
Bases: pybrain.rl.explorers.discrete.discrete.DiscreteExplorer

A discrete explorer, that directly manipulates the ActionValue estimator (table or network) and keeps the changes fixed for one full episode (if episodic) or slowly changes it over time.

TODO: currently only implemented for episodes

activate(*state, action*)
Save the current state for state-dependent exploration.

_forwardImplementation(*inbuf, outbuf*)
Activate the copied module instead of the original and feed it with the current state.

5.9 learners – RL Components: Learners

5.9.1 Abstract classes

The top of the learner hierarchy is more conceptual than functional. The different classes distinguish algorithms in such a way that we can automatically determine when an algorithm is not applicable for a problem.

class **Learner**()
Top-level class for all reinforcement learning algorithms. Any learning algorithm changes a policy (in some way) in order to increase the expected reward/fitness.

learn()
The main method, that invokes a learning step.

class **EpisodicLearner**()
Bases: pybrain.rl.learners.learner.Learner

Assumes the task is episodic, not life-long, and therefore does a learning step only after the end of each episode.

class `DataSetLearner`()
Bases: `pybrain.rl.learners.learner.EpisodicLearner`

A class for learners that learn from a dataset, which has no target output but only a reinforcement signal for each sample. It requires a ReinforcementDataSet object (which provides state-action-reward tuples).

class `ExploringLearner`()
Bases: `pybrain.rl.learners.learner.Learner`

A Learner determines how to change the adaptive parameters of a module.

class `DirectSearchLearner`()
Bases: `pybrain.rl.learners.learner.Learner`

The class of learners that (in contrast to value-based learners) searches directly in policy space.

5.9.2 Value-based Learners

class `ValueBasedLearner`()
Bases: `pybrain.rl.learners.learner.ExploringLearner`, `pybrain.rl.learners.learner.DataSetLe`
`pybrain.rl.learners.learner.EpisodicLearner`

An RL algorithm based on estimating a value-function.

batchMode
Does the algorithm run in batch mode or online?

explorer
Return the internal explorer.

module
Return the internal module.

offPolicy
Does the algorithm work on-policy or off-policy?

class `Q`(*alpha=0.5, gamma=0.98999999999999999*)
Bases: `pybrain.rl.learners.valuebased.valuebased.ValueBasedLearner`

learn()
Learn on the current dataset, either for many timesteps and even episodes (batchMode = True) or for a single timestep (batchMode = False). Batch mode is possible, because Q-Learning is an off-policy method.

In batchMode, the algorithm goes through all the samples in the history and performs an update on each of them. if batchMode is False, only the last data sample is considered. The user himself has to make sure to keep the dataset consistent with the agent's history.

class `QLambda`(*alpha=0.5, gamma=0.98999999999999999, qlambda=0.90000000000000002*)
Bases: `pybrain.rl.learners.valuebased.valuebased.ValueBasedLearner`

Q-lambda is a variation of Q-learning that uses an eligibility trace.

class `SARSA`(*alpha=0.5, gamma=0.98999999999999999*)
Bases: `pybrain.rl.learners.valuebased.valuebased.ValueBasedLearner`

State-Action-Reward-State-Action (SARSA) algorithm.

In batchMode, the algorithm goes through all the samples in the history and performs an update on each of them.

class NFQ( )
Bases: pybrain.rl.learners.valuebased.valuebased.ValueBasedLearner

Neuro-fitted Q-learning

5.9.3 Direct-search Learners

class PolicyGradientLearner( )
Bases: pybrain.rl.learners.directsearch.directsearch.DirectSearchLearner, pybrain.rl.learners.learner.DataSetLearner, pybrain.rl.learners.learner.ExploringLearner

PolicyGradientLearner is a super class for all continuous direct search algorithms that use the log likelihood of the executed action to update the weights. Subclasses are ENAC, GPOMDP, or REINFORCE.

learn( )
calls the gradient calculation function and executes a step in direction of the gradient, scaled with a small learning rate alpha.

class Reinforce( )
Bases: pybrain.rl.learners.directsearch.policygradient.PolicyGradientLearner

Reinforce is a gradient estimator technique by Williams (see "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning"). It uses optimal baselines and calculates the gradient with the log likelihoods of the taken actions.

class ENAC( )
Bases: pybrain.rl.learners.directsearch.policygradient.PolicyGradientLearner

Episodic Natural Actor-Critic. See J. Peters "Natural Actor-Critic", 2005. Estimates natural gradient with regression of log likelihoods to rewards.

Note: Black-box optimization algorithms can also be seen as direct-search RL algorithms, but are not included here.

5.10 tasks – RL Components: Tasks

class Task(*environment*)
A task is associating a purpose with an environment. It decides how to evaluate the observations, potentially returning reinforcement rewards or fitness values. Furthermore it is a filter for what should be visible to the agent. Also, it can potentially act as a filter on how actions are transmitted to the environment.

denormalize(*actors*)
The function scales the parameters from -1 and 1 to the given interval (min, max) for each actor.

getObservation( )
A filtered mapping to getSample of the underlying environment.

getReward( )
Compute and return the current reward (i.e. corresponding to the last action performed)

normalize(*sensors*)
The function scales the parameters to be between -1 and 1. e.g. [(-pi, pi), (0, 1), (-0.001, 0.001)]

performAction(*action*)
A filtered mapping towards performAction of the underlying environment.

setScaling(*sensor_limits, actor_limits*)
Expects scaling lists of 2-tuples - e.g. [(-3.14, 3.14), (0, 1), (-0.001, 0.001)] - one tuple per parameter,

giving min and max for that parameter. The functions normalize and denormalize scale the parameters between -1 and 1 and vice versa. To disable this feature, use 'None'.

class **EpisodicTask**(*environment*)
Bases: pybrain.rl.environments.task.Task, pybrain.rl.environments.fitnessevaluator.Fitness

A task that consists of independent episodes.

addReward()
A filtered mapping towards performAction of the underlying environment.

discount
Discount factor

f(x)
An episodic task can be used as an evaluation function of a module that produces actions from observations, or as an evaluator of an agent.

getTotalReward()
Return the accumulated reward since the start of the episode

isFinished()
Is the current episode over?

performAction(*action*)
Execute one action.

reset()
Re-initialize the environment

5.11 optimization – Black-box Optimization Algorithms

5.11.1 The two base classes

class **BlackBoxOptimizer**(*evaluator=None, initEvaluable=None, **kwargs*)
The super-class for learning algorithms that treat the problem as a black box. At each step they change the policy, and get a fitness value by invoking the FitnessEvaluator (provided as first argument upon initialization).

Evaluable objects can be lists or arrays of continuous values (also wrapped in ParameterContainer) or subclasses of Evolvable (that define its methods).

__init__(*evaluator=None, initEvaluable=None, **kwargs*)
The evaluator is any callable object (e.g. a lambda function). Algorithm parameters can be set here if provided as keyword arguments.

setEvaluator(*evaluator, initEvaluable=None*)
If not provided upon construction, the objective function can be given through this method. If necessary, also provide an initial evaluable.

learn(*additionalLearningSteps=None*)
The main loop that does the learning.

minimize
Minimize cost or maximize fitness? By default, all functions are maximized.

maxEvaluations
Stopping criterion based on number of evaluations.

maxLearningSteps
Stopping criterion based on number of learning steps.

desiredEvaluation
Is there a known value of sufficient fitness?

verbose
provide console output during learning

storeAllEvaluations
Store all evaluations (in the ._allEvaluations list)?

storeAllEvaluated
Store all evaluated instances (in the ._allEvaluated list)?

numParameters
dimension of the search space, if applicable

class **ContinuousOptimizer**(*evaluator=None, initEvaluable=None, **kwargs*)
Bases:

class **FEM**(*evaluator=None, initEvaluable=None, **kwargs*)
Fitness Expectation-Maximization (PPSN 2008).

Finite difference methods

class **FiniteDifferences**(*evaluator=None, initEvaluable=None, **kwargs*)
Basic finite difference method.

perturbation()
produce a parameter perturbation

class **PGPE**(*evaluator=None, initEvaluable=None, **kwargs*)
Bases: `pybrain.optimization.finitedifference.fd.FiniteDifferences`

Policy Gradients with Parameter Exploration (ICANN 2008).

epsilon
Initial value of sigmas

exploration
exploration type

learningRate
specific settings for sigma updates

momentum
momentum term (0 to deactivate)

perturbation()
Generate a difference vector with the given standard deviations

rprop
rprop decent (False to deactivate)

sigmaLearningRate
specific settings for sigma updates

wDecay
lasso weight decay (0 to deactivate)

class **SimpleSPSA**(*evaluator=None, initEvaluable=None, **kwargs*)
Bases: `pybrain.optimization.finitedifference.fd.FiniteDifferences`

Simultaneous Perturbation Stochastic Approximation.

This class uses SPSA in general, but uses the likelihood gradient and a simpler exploration decay.

Population-based

class **ParticleSwarmOptimizer**(*evaluator=None, initEvaluable=None, **kwargs*)
Particle Swarm Optimization

size determines the number of particles.

boundaries should be a list of (min, max) pairs with the length of the dimensionality of the vector to be optimized (default: +-10). Particles will be initialized with a position drawn uniformly in that interval.

memory indicates how much the velocity of a particle is affected by its previous best position.

sociality indicates how much the velocity of a particle is affected by its neighbours best position.

inertia is a damping factor.

best(*particlelist*)
Return the particle with the best fitness from a list of particles.

class GA(*evaluator=None, initEvaluable=None, **kwargs*)
Standard Genetic Algorithm.

crossOver(*parents, nbChildren*)
generate a number of children by doing 1-point cross-over

mutated(*indiv*)
mutate some genes of the given individual

mutationProb
mutation probability

produceOffspring()
produce offspring by selection, mutation and crossover.

select()
select some of the individuals of the population, taking into account their fitnesses

Returns list of selected parents

selectionSize
the number of parents selected from the current population

topProportion
selection proportion

tournament
selection scheme

5.11.4 Multi-objective Optimization

class MultiObjectiveGA(*evaluator=None, initEvaluable=None, **kwargs*)
Multi-objective Genetic Algorithm: the fitness is a vector with one entry per objective. By default we use NSGA-II selection.

5.12 `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 explicity 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 explicity by *nb_classes*. To give the classes names, supply an iterable of strings as *class_labels*.

5.13 importance – Datasets for Weighted Supervised Training

class ImportanceDataSet(*indim, targetdim*)
Bases: pybrain.datasets.sequential.SequentialDataSet

Allows setting an importance value for each of the targets of a sample.

addSample(*inp, target, importance=None*)
adds a new sample consisting of input, target and importance.

Parameters

- *inp* – the input of the sample
- *target* – the target of the sample

Key importance the importance of the sample. If left None, the importance will be set to 1.0

5.14 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.

5.15 `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.

class **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.

class **reconstruct**(*filename*)
Read an incomplete data set (option arraysonly) 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.

5.16 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, 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.

Parameter *dataset* – A non-sequential supervised data set.

Key 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

5.17 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.

Key search optional name of grid search class to use for RBF kernels: 'GridSearch' or 'GridSearchDOE'

Key log2g base 2 log of the RBF width parameter

Key log2c base 2 log of the slack parameter

Key searchlog filename into which to dump the search log

Key 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.

Key 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.

Parameters

- *problem* – the LIBSVM svm_problem to be optimized, ie. the input and target data
- *targets* – unfortunately, the targets used in the problem definition have to be given again here
- *cmin* – lower left corner of the log2C/log2gamma window to search
- *cmax* – upper right corner of the log2C/log2gamma window to search

Key cstep step width for log2C and log2gamma (ignored for DOE search)

Key crossval split dataset into this many parts for cross-validation

Key plotflag if True, plot the error surface contour (regular) or search pattern (DOE)

Key maxdepth maximum window bisection depth (DOE only)

Key searchlog Save a list of coordinates and the achieved CV accuracy to this file

Key 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

5.18 `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 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=9.9999999999999995e-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=9.9999999999999995e-07, deltamax=5.0, delta0=0.10000000000000001, **kwargs*)
Set up training algorithm parameters, and objects associated with the trainer.

Parameter *module* – the module whose parameters should be trained.

Key etaminus factor by which step width is decreased when overstepping (0.5)

Key etaplus factor by which step width is increased when following gradient (1.2)

Key delta step width for each weight

Key deltamin minimum step width (1e-6)

Key deltamax maximum step width (5.0)

Key delta0 initial step width (0.1)

Note: See the documentation of `BackpropTrainer` for inherited methods.

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.

Parameters

- *DSseq* – the sequential data set to cut up
- *winsize* – size of the data window
- *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.

class **ESS**(*output, target*)
Returns the explained sum of squares (ESS).

Parameters

- *output* – array of output values
- *target* – array of target values

class **MSE**(*output, target, importance=None*)
Returns the mean squared error. The multidimensional arrays will get flattened in order to compare them.

Parameters

- *output* – array of output values
- *target* – array of target values

Key importance each squared error will be multiplied with its corresponding importance value. After summing up these values, the result will be divided by the sum of all importance values for normalization purposes.

class **classificationPerformance**(*output, target*)
Returns the hit rate of the outputs compared to the targets.

Parameters

- *output* – array of output values
- *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.

class **MSE**(*module, dataset*)
Returns the mean squared error.

Parameters

- *module* – Object of any subclass of pybrain's Module type
- *dataset* – Dataset object at least containing the fields 'input' and 'target' (for example SupervisedDataSet)

class **calculateModuleOutput**(*module, dataset*)
Calculates the module's output on the dataset. Can be called with any type of dataset.

Parameter *dataset* – Any Dataset object containing an 'input' field.

class **classificationPerformance**(*module, dataset*)
Returns the hit rate of the module's output compared to the targets stored inside dataset.

Parameters

- *module* – Object of any subclass of pybrain's Module type
- *dataset* – Dataset object at least containing the fields 'input' and 'target' (for example SupervisedDataSet)

class **validate**(*valfunc, module, dataset*)
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.

Parameters

- *valfunc* – A function expecting arrays for output, target and importance (optional). See Validator.MSE for an example.
- *module* – Object of any subclass of pybrain's Module type
- *dataset* – Dataset object at least containing the fields '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 ist 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 the mean of the calculated validation results will be returned.

setArgs(***kwargs*)
Set the specified member variables.

Key max_epochs maximum number of epochs the trainer should train the module for.

Key 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.

Auxiliary functions

semilinear(*x*)
This function ensures that the values of the array are always positive. It is x+1 for x=>0 and exp(x) for x<0.

semilinearPrime(*x*)
This function is the first derivative of the semilinear function (above). It is needed for the backward pass of the module.

sigmoid(*x*)
Logistic sigmoid function.

sigmoidPrime(*x*)
Derivative of logistic sigmoid.

tanhPrime(*x*)
Derivative of tanh.

safeExp(*x*)
Bounded range for the exponential function (won't rpoduce inf or NaN).

ranking(*R*)
Produces a linear ranking of the values in R.

multivariateNormalPdf(*z, x, sigma*)
The pdf of a multivariate normal distribution (not in scipy). The sample z and the mean x should be 1-dim-arrays, and sigma a square 2-dim-array.

simpleMultivariateNormalPdf(*z, detFactorSigma*)
Assuming z has been transformed to a mean of zero and an identity matrix of covariances. Needs to provide the determinant of the factorized (real) covariance matrix.

multivariateCauchy(*mu, sigma, onlyDiagonal=True*)
Generates a sample according to a given multivariate Cauchy distribution.

5.20 `utilities` – Simple but useful Utility Functions

Tools for binary encodings

one_to_n(*val, maxval*)
Returns a 1-in-n binary encoding of a non-negative integer.

n_to_one(*arr*)
Returns the reverse of a 1-in-n binary encoding.

int2gray(*i*)
Returns the value of an integer in Gray encoding.

asBinary(*i*)
Produces a string from an integer's binary representation. (preceding zeros removed).

Tools for sets

reachable(*stepFunction, start, destinations*)
Determines the subset of destinations that can be reached from a set of starting positions, while using stepFunction (which produces a list of neighbor states) to navigate. Uses breadth-first search.

crossproduct(*ss, row=None, level=0*)
Returns the cross-product of the sets given in *ss*.

Matrix tools

triu2flat(*m*)
Flattens an upper triangular matrix, returning a vector of the non-zero elements.

flat2triu(*a, dim*)
Produces an upper triangular matrix of dimension dim from the elements of the given vector.

blockList2Matrix(*l*)
Converts a list of matrices into a corresponding big block-diagonal one.

blockCombine(*l*)
Produce a matrix from a list of lists of its components.

Stochastic index choices

drawIndex(*probs, tolerant=False*)
Draws an index given an array of probabilities.

Key tolerant if set to True, the array is normalized to sum to 1.

drawGibbs(*vals, temperature=1.0*)
Return the index of the sample drawn by a softmax (Gibbs).

5.21 nearoptimal – Near Optimal Locality Sensitive Hashing

class **MultiDimHash**(*dim, omega=4, prob=0.80000000000000004*)
Class that represents a datastructure that enables nearest neighbours search and methods to do so.

__init__(*dim, omega=4, prob=0.80000000000000004*)
Create a hash for arrays of dimension dim.

The hyperspace will be split into hypercubes with a sidelength of omega * sqrt(sqrt(dim)), that is omega * radius.

Every point in the dim-dimensional euclidean space will be hashed to its correct bucket with a probability of prob.

insert(*point, satellite*)
Put a point and its satellite information into the hash structure.

knn(*point, k*)
Return the k approximate nearest neighbours of the item in the current hash.

Mind that the probabilistic nature of the data structure might not return a nearest neighbor at all and not the nearest neighbour.

See Also:

Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions Paper that describes the algorithm used in this module by Alexandr Andoni and Piotr Indyk.

CHAPTER

SIX

INDICES AND TABLES

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

MODULE INDEX

P

INDEX

Symbols

A

B

C

D

F

G

H

I

Q

R

S

T

V

W