Modular Toolkit for Data Processing (MDP)

This document is also available online: http://mdp-toolkit.sourceforge.net/tutorial.html

Tutorial

Author: Pietro Berkes and Tiziano Zito

Homepage: http://mdp-toolkit.sourceforge.net

Version: 2.0.0

This is a guide to basic and some more advanced features of the MDP library. Besides the present tutorial, you can learn more about MDP by using the standard Python tools. All MDP nodes have doc-strings, the public attributes and methods have telling names: All information about a node can be obtained using the `help` and `dir` functions within the Python interpreter. In addition to that, an automatically generated API is available.

Note

Code snippets throughout the script will be denoted by:

```
>>> print "Hello world!"
Hello world!
```

To run the following code examples don't forget to import mdp in your Python session with:

```
>>> import mdp
```

You'll find all the code of this tutorial within the `demo` directory in the MDP installation path.

Contents

Introduction

Modular toolkit for Data Processing (MDP) is a Python library to perform data processing. Already implemented algorithms include: Principal Component Analysis (PCA), Independent Component Analysis (ICA), Slow Feature Analysis (SFA), and Growing Neural Gas (GNG).

MDP supports the most common numerical extensions to Python and the symeig package (a Python wrapper for the LAPACK functions to solve the standard and generalized eigenvalue problems for symmetric (hermitian) positive definite matrices). MDP also includes `graph` (a lightweight package to handle graphs).

When used together with SciPy (the scientific Python library) and symeig, MDP gives to the scientific programmer the full power of well-known C and FORTRAN data processing libraries. MDP helps the programmer to exploit Python object oriented design with C and FORTRAN efficiency.

MDP has been written for research in neuroscience, but it has been designed to be helpful in any context where trainable data processing algorithms are used. Its simplicity on the user side together with the reusability of the implemented nodes could make it also a valid educational tool.

Quick Start

Using MDP is as easy as:

```
>>> import mdp
>>> # perform pca on some data x
...
>>> y = mdp.pca(x)
>>> # perform ica on some data x using single precision
...
>>> y = mdp.fastica(x, dtype='f')
```

A complete list of all short-cut functions like `pca` or `fastica` can be obtained as follows:

```
>>> dir(mdp.helper_funcs)
['__builtins__', '__doc__', '__file__', '__name__',
 'cubica', 'fastica', 'get_eta', 'mdp', 'pca', 'sfa', 'whitening']
```

MDP is of course much more than this: it allows to combine different algorithms and other data processing elements (nodes) into data processing sequences (flows). Moreover, it provides a framework that makes the implementation of new algorithms easy and intuitive.

Nodes

A node is the basic unit in MDP and it represents a data processing element, like for example a learning algorithm, a filter, a visualization step etc. Each node can have a training phase, during which the internal structures are learned from training data (e.g. the weights of a neural network are adapted or the covariance matrix is estimated) and an execution phase, where new data can be processed forwards (by processing the data through the node) or backwards (by applying the inverse of the transformation computed by the node if defined). MDP is designed to make the implementation of new algorithms easy and intuitive, for example by setting automatically input and output dimension and by casting the data to match the `dtype` (e.g. float or double precision) of the internal structures. Most of the nodes were designed to be applied to arbitrarily long sets of data: the internal structures can be updated successively by sending chunks of the input data (this is equivalent to online learning if the chunks consists of single observations, or to batch learning if the whole data is sent in a single chunk). Already implemented nodes include Principal Component Analysis (PCA), Independent Component Analysis (ICA), Slow Feature Analysis (SFA), and Growing Neural Gas Network. Have a look at the full list of implemented nodes.

Node Instantiation

Nodes can be obtained by creating an instance of the `Node` class. Each node is characterized by an input dimension, that corresponds to the dimensionality of the input vectors, an output dimension, and a `dtype`, which determines the numerical type of the internal structures and of the output signal. These three attributes are inherited from the input data if left unspecified. Input dimension and `dtype` can usually be specified when an instance of the node class is created. The constructor of each node class can require other task-specific arguments. The full documentation is available in the doc-string of the node's class.

Some examples of node instantiation:

- Create a node that performs Principal Component Analysis (PCA) whose input dimension and `dtype` are inherited from the input data during training. Output dimensions default to input dimensions.

```
>>> pcanode1 = mdp.nodes.PCANode()
>>> pcanode1
PCANode(input_dim=None, output_dim=None, dtype=None)
```

- Setting `output_dim = 10` means that the node will keep only the first 10 principal components of the input.

```
>>> pcanode2 = mdp.nodes.PCANode(output_dim = 10)
>>> pcanode2
PCANode(input_dim=None, output_dim=10, dtype=None)
```

  The output dimensionality can also be specified in terms of the explained variance. If we want to keep the number of principal components which can account for 80% of the input variance, we set:

```
>>> pcanode3 = mdp.nodes.PCANode(output_dim = 0.8)
>>> pcanode3.desired_variance
0.80000000000000004
```

- If `dtype` is set to `f` (32-bit float), the input data is cast to single precision when received and the internal structures are also stored as `f`. `dtype` influences the memory space necessary for a node and the precision with which the computations are performed.

```
>>> pcanode4 = mdp.nodes.PCANode(dtype = 'f')
>>> pcanode4
PCANode(input_dim=None, output_dim=None, dtype='f')
```

You can obtain a list of the numerical types supported by a node by calling its `get_supported_dtypes` method:

```
>>> pcanode4.get_supported_dtypes()
[dtype('<f4'), dtype('<f8')]
```

This method returns a list of `numpy.dtype` objects (see the `numpy` documentation for more details.

- A `PolynomialExpansionNode` expands its input in the space of polynomals of a given degree by computing all monomials up to the specified degree. Its constructor needs as first argument the degree of the polynomials space (3 in this case).

```
>>> expnode = mdp.nodes.PolynomialExpansionNode(3)
```

Node Training

Some nodes need to be trained to perform their task. This can be done during a training phases by calling the `train` method. MDP supports both supervised and unsupervised training, and algorithms with multiple training phases.

Some examples of node training:

- Create some random data and update the internal structures (i.e. mean and covariance matrix) of the `PCANode`:

```
>>> x = mdp.numx_rand.random((100, 25))  # 25 variables, 100 observa-
tions
>>> pcanode1.train(x)
```

At this point the input dimension and the `dtype` have been inherited from `x`:

```
>>> pcanode1
PCANode(input_dim=25, output_dim=None, dtype='float64')
```

- We can train our node with more than one chunk of data. This is especially useful when the input data is too long to be stored in memory or when it has to be created on-the-fly. (See also the Iterators section):

```
>>> for i in range(100):
...     x = mdp.numx_rand.random((100, 25))
...     pcanode1.train(x)
>>>
```

- Some nodes don't need to or cannot be trained:

```
>>> expnode.is_trainable()
False
```

Trying to train them anyway would raise an `IsNotTrainableException`.

- The training phase ends when the `stop_training`, `execute`, `inverse`, and possibly some other node-specific methods are called. For example we can stop the training of `pcanode1` (at this point the principal components are computed):

```
>>> pcanode1.stop_training()
```

- If the `PCANode` was declared to have a number of output components dependent on the input variance to be explained, we can check after training the number of output components and the actually explained variance:

```
>>> pcanode3.train(x)
>>> pcanode3.stop_training()
>>> pcanode3.output_dim
16
>>> pcanode3.explained_variance
0.85261144755506446
```

  It is now possible to access the trained internal data. In general, a list of the interesting internal attributes can be found in the class documentation.

```
>>> avg = pcanode1.avg            # mean of the input data
>>> v = pcanode1.get_projmatrix() # projection matrix
```

- Some nodes, namely the one corresponding to supervised algorithms, e.g. Fisher Discriminant Analysis (FDA), may need some labels or other supervised signals to be passed during training. Detailed information about the signature of the `train` method can be read in its doc-string.

```
>>> fdanode = mdp.nodes.FDANode()
>>> for label in ['a', 'b', 'c']:
...     x = mdp.numx_rand.random((100, 25))
...     fdanode.train(x, label)
>>>
```

- A node could also require multiple training phases. For example, the training of `fdanode` is not complete yet, since it has two training phases. We need to stop the first phase and train the second:

```
>>> fdanode.stop_training()
>>> for label in ['a', 'b', 'c']:
...     x = mdp.numx_rand.random((100, 25))
...     fdanode.train(x, label)
>>>
```

  The easiest way to train multiple phase nodes is using Flows , which automatically handle multiple phases.

Node Execution

After the training phase it is possible to execute the node:

- The input data is projected on the principal components learned in the training phase:

```
>>> x = mdp.numx_rand.random((100, 25))
>>> y_pca = pcanode1.execute(x)
```

- Calling a node instance is equivalent to executing it:

```
>>> y_pca = pcanode1(x)
```

- The input data is expanded in the space of polynomials of degree 3:

```
>>> x = mdp.numx_rand.random((100, 5))
>>> y_exp = expnode(x)
```

- The input data is projected to the directions learned by FDA:

```
>>> x = mdp.numx_rand.random((100, 25))
>>> y_fda = fdanode(x)
```

- Some nodes may allow for optional arguments in the `execute` method, as always the complete information is given in the doc-string.

Node Inversion

If the operation computed by the node is invertible, it is possible to compute the inverse transformation:

- Given the output data, compute the inverse projection to the input space for the PCA node:

```
>>> pcanode1.is_invertible()
True
>>> x = pcanode1.inverse(y_pca)
```

- The expansion node in not invertible:

```
>>> expnode.is_invertible()
False
```

  Trying to compute the inverse would raise an `IsNotInvertibleException`.

Writing your own nodes: subclassing Node

MDP tries to make it easy to write new data processing elements that fit with the existing elements. To expand the MDP library of implemented nodes with your own nodes you can subclass the Node class, overriding some of the methods according to your needs.

We'll illustrate this with some toy examples.

- We start by defining a node that multiplies its input by 2.

  Define the class as a subclass of Node:

```
>>> class TimesTwoNode(mdp.Node):
```

  This node cannot be trained. To specify this, one has to overwrite the `is_trainable` method to return False:

```
...     def is_trainable(self): return False
```

  Execute only needs to multiply x by 2

```
...     def _execute(self, x):
...         return 2*x
```

  Note that the `execute` method, which should never be overwritten and which is inherited from the `Node` parent class, will perform some tests, for example to make sure that `x` has the right rank, dimensionality and casts it to have the right `dtype`. After that the user-supplied `_execute` method is called. Each subclass has to handle the `dtype` defined by the user or inherited by the input data, and make sure that internal structures are stored consistently. To help with this the `Node` base class has a method called `_refcast(array, dtype)` that casts an array only when its `dtype` is different from the requested one.

  The inverse of the multiplication by 2 is of course the division by 2:

```
...     def _inverse(self, y):
...         return y/2
...
>>>
```

Test the new node:

```
>>> node = TimesTwoNode(dtype = 'i')
>>> x = mdp.numx.array([[1.0, 2.0, 3.0]])
>>> y = node(x)
>>> print x, '* 2 =  ', y
[ [ 1.  2.  3.]] * 2 =   [ [2 4 6]]
>>> print y, '/ 2 =', node.inverse(y)
[ [2 4 6]] / 2 = [ [1 2 3]]
```

- We then define a node that raises the input to the power specified in the initializer:

```
>>> class PowerNode(mdp.Node):
```

We redefine the init method to take the power as first argument. In general one should always give the possibility to set the `dtype` and the input dimensions. The default value is `None`, which means that the exact value is going to be inherited from the input data:

```
...     def __init__(self, power, input_dim=None, dtype=None):
```

Initialize the parent class:

```
...         super(PowerNode, self).__init__(input_dim=input_dim, dtype=dtype)
```

Store the power:

```
...         self.power = power
```

`PowerNode` is not trainable...

```
...     def is_trainable(self): return False
```

... nor invertible:

```
...     def is_invertible(self): return False
```

It is possible to overwrite the function `_get_supported_dtypes` to return a list of `dtype` supported by the node:

```
...     def _get_supported_dtypes(self):
...         return ['f', 'd']
```

The supported types can be specified in any format allowed by `numpy.dtype`. The interface method `get_supported_dtypes` converts them and returns a list of `dtype` objects.

The `_execute` method:

```
...     def _execute(self, x):
...         return self._refcast(x**self.power)
...
>>>
```

Test the new node

```
>>> node = PowerNode(3)
>>> x = mdp.numx.array([[1.0, 2.0, 3.0]])
>>> y = node.execute(x)
>>> print x, '**', node.power, '=', node(x)
[ [ 1.  2.  3.]] ** 3 = [ [  1.   8.  27.]]
```

- We now define a node that needs to be trained. The `MeanFreeNode` computes the mean of its training data and subtracts it from the input during execution:

```
>>> class MeanFreeNode(mdp.Node):
...     def __init__(self, input_dim=None, dtype=None):
...         super(MeanFreeNode, self).__init__(input_dim=input_dim,
...                                            dtype=dtype)
```

We store the mean of the input data in an attribute. We initialize it to `None` since we still don't know how large is an input vector:

```
...         self.avg = None
```

Same for the number of training points:

```
...         self.tlen = 0
```

The subclass only needs to overwrite the `_train` method, which will be called by the parent `train` after some testing and casting has been done:

```
...     def _train(self, x):
...         # Initialize the mean vector with the right
...         # size and dtype if necessary:
...         if self.avg is None:
...             self.avg = mdp.numx.zeros(self.input_dim,
...                                       dtype=self.dtype)
```

Update the mean with the sum of the new data:

```
...         self.avg += mdp.numx.sum(x, axis=0)
```

Count the number of points processed:

```
...         self.tlen += x.shape[0]
```

Note that `train` method can have further arguments, which might be useful to implement algorithms that require supervised learning. For example, if you want to define a node that performs some form of classification you can define a `_train(self, data, labels)` method. The parent `train` checks `data` and takes care to pass the `labels` on (cf. for example `mdp.nodes.FDANode`).

The `_stop_training` function is called by the parent `stop_training` method when the training phase is over. We divide the sum of the training data by the number of training vectors to obtain the mean:

```
...     def _stop_training(self):
...         self.avg /= self.tlen
```

The `_execute` and `_inverse` methods:

```
...     def _execute(self, x):
...         return x - self.avg
...     def _inverse(self, y):
...         return y + self.avg
...
>>>
```

Test the new node:

```
>>> node = MeanFreeNode()
>>> x = mdp.numx_rand.random((10,4))
>>> node.train(x)
>>> y = node.execute(x)
>>> print 'Mean of y (should be zero): ', mdp.numx.mean(y, 0)
Mean of y (should be zero):  [  0.00000000e+00   2.22044605e-17
-2.22044605e-17   1.11022302e-17]
```

- It is also possible to define nodes with multiple training phases. In such a case, calling the `train` and `stop_training` functions multiple times is going to execute successive training phases (this kind of node is much easier to train using Flows). Here we'll define a node that returns a meanfree, unit variance signal. We define two training phases: first we compute the mean of the signal and next we sum the squared, meanfree input to compute the standard deviation (of course it is possible to solve this problem in one single step - remeber this is just a toy example).

```
>>> class UnitVarianceNode(mdp.Node):
...     def __init__(self, input_dim=None, dtype=None):
...         super(UnitVarianceNode, self).__init__(input_dim=input_dim,
...                                                dtype=dtype)
...         self.avg = None # average
...         self.std = None # standard deviation
...         self.tlen = 0
```

The training sequence is defined by the user-supplied function `_get_train_seq`, that returns a list of tuples, one for each training phase. The tuples contain references to the training and stop-training functions of each of them. The default output of this function is `[(_train, _stop_training)]`, which explains the standard behavior illustrated above. We overwrite the function to return the list of our training functions:

```
...     def _get_train_seq(self):
...         return [(self._train_mean, self._stop_mean),
...                 (self._train_std, self._stop_std)]
```

Next we define the training functions. The first phase is identical to the one in the previous example:

```
...     def _train_mean(self, x):
...         if self.avg is None:
...             self.avg = mdp.numx.zeros(self.input_dim,
...                                       dtype=self.dtype)
...         self.avg += mdp.numx.sum(x, 0)
...         self.tlen += x.shape[0]
...     def _stop_mean(self):
...         self.avg /= self.tlen
```

The second one is only marginally different and does not require many explanations:

```
...     def _train_std(self, x):
...         if self.std is None:
...             self.tlen = 0
...             self.std = mdp.numx.zeros(self.input_dim,
...                                       dtype=self.dtype)
...         self.std += mdp.numx.sum((x - self.avg)**2., 0)
...         self.tlen += x.shape[0]
```

```
...     def _stop_std(self):
...         # compute the standard deviation
...         self.std = mdp.numx.sqrt(self.std/(self.tlen-1))
```

The `_execute` and `_inverse` methods are not surprising, either:

```
...     def _execute(self, x):
...         return (x - self.avg)/self.std
...     def _inverse(self, y):
...         return y*self.std + self.avg
>>>
```

Test the new node:

```
>>> node = UnitVarianceNode()
>>> x = mdp.numx_rand.random((10,4))
>>> # loop over phases
... for phase in range(2):
...     node.train(x)
...     node.stop_training()
...
...
>>> # execute
... y = node.execute(x)
>>> print 'Standard devia-
tion of y (should be one): ', mdp.numx.std(y, 0)
Standard deviation of y (should be one):  [ 1.  1.  1.  1.]
```

- In our last example we'll define a node that returns two copies of its input. The output is going to have twice as many dimensions.

```
>>> class TwiceNode(mdp.Node):
...     def is_trainable(self): return False
...     def is_invertible(self): return False
```

When `Node` inherits the input dimension, output dimension, and `dtype` from the input data, it calls the methods `set_input_dim`, `set_output_dim`, and `set_dtype`. Those are the setters for `input_dim`, `output_dim` and `dtype`, which are Python properties. If a subclass needs to change the default behaviour, the internal methods `_set_input_dim`, `_set_output_dim` and `_set_dtype` can be overwritten. The property setter will call the internal method after some basic testing and internal settings. The private methods `_set_input_dim`, `_set_output_dim` and `_set_dtype` are responsible for setting the private attributes `_input_dim`, `_output_dim`, and `_dtype` that contain the actual value.

Here we overwrite `_set_input_dim` to automatically set the output dimension to be twice the input one, and `_set_output_dim` to raise an exception, since the output dimension should not be set explicitly.

```
...     def _set_input_dim(self, n):
...         self._input_dim = n
...         self._output_dim = 2*n
...     def _set_output_dim(self, n):
...         raise mdp.NodeException, "Output dim can not be set explicitly!"
```

The `_execute` method:

```
...     def _execute(self, x):
...         return mdp.numx.concatenate((x, x), 1)
...
>>>
```

Test the new node

```
>>> node = TwiceNode()
>>> x = mdp.numx.zeros((5,2))
>>> x
array([[0, 0],
       [0, 0],
       [0, 0],
       [0, 0],
       [0, 0]])
>>> node.execute(x)
array([[0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]])
```

Flows

A flow consists in an acyclic graph of nodes (currently only node sequences are implemented). The data is sent to an input node and is successively processed by the following nodes on the graph. The general flow implementation automatizes the training, execution, and inverse execution (if defined) of the whole graph. Crash recovery is optionally available: in case of failure the current state of the flow is saved for later inspection. A subclass of the basic flow class allows user-supplied checkpoint functions to be executed at the end of each phase, for example to save the internal structures of a node for later analysis.

Flow instantiation, training and execution

Suppose we have an input signal with an high number of dimensions, on which we would like to perform ICA. To make the problem affordable, we first need to reduce its dimensionality with PCA. Finally, we would like to find out the data that produces local maxima in the output on a new test set. This information could be used to characterize the input-output filters.

We start by generating some input signal at random (which makes the example useless, but it's just for illustration...). Generate 1000 observations of 20 independent source signals:

```
>>> inp = mdp.numx_rand.random((1000, 20))
```

Rescale x to have zero mean and unit variance:

```
>>> inp = (inp - mdp.numx.mean(inp, 0))/mdp.numx.std(inp, 0)
```

We reduce the variance of the last 15 components, so that they are going to be eliminated by PCA:

```
>>> inp[:,5:] /= 10.0
```

Mix the input signals linearly:

```
>>> x = mdp.utils.mult(inp,mdp.numx_rand.random((20, 20)))
```

x is now the training data for our simulation. In the same way we also create a test set x_test.

```
>>> inp_test = mdp.numx_rand.random((1000, 20))
>>> inp_test = (inp_test -
mdp.numx.mean(inp_test, 0))/mdp.numx.std(inp_test, 0)
>>> inp_test[:,5:] /= 10.0
>>> x_test = mdp.utils.mult(inp_test, mdp.numx_rand.random((20, 20)))
```

- We could now perform our analysis using only nodes, that's the lenghty way...

  1. Perform PCA:

     ```
     >>> pca = mdp.nodes.PCANode(output_dim=5)
     >>> pca.train(x)
     >>> out1 = pca.execute(x)
     ```

  2. Perform ICA using CuBICA algorithm:

     ```
     >>> ica = mdp.nodes.CuBICANode()
     >>> ica.train(out1)
     >>> out2 = ica.execute(out1)
     ```

  3. Find the three largest local maxima in the output of the ICA node when applied to the test data, using a `HitParadeNode`:

     ```
     >>> out1_test = pca.execute(x_test)
     >>> out2_test = ica.execute(out1_test)
     >>> hitnode = mdp.nodes.HitParadeNode(3)
     >>> hitnode.train(out2_test)
     >>> maxima, indices = hitnode.get_maxima()
     ```

- ... or we could use flows, which is the best way:

  ```
  >>> flow = mdp.Flow([mdp.nodes.PCANode(output_dim=5), mdp.nodes.CuBICANode()])
  >>> flow.train(x)
  ```

  You will probably get some warnings here. This is expected, see the section about Iterators to learn more about that, for the moment you can simply ignore them.

  Now the training phase of PCA and ICA are completed. Next we append a `HitParadeNode` which we want to train on the test data:

  ```
  >>> flow.append(mdp.nodes.HitParadeNode(3))
  >>> flow.train(x_test)
  >>> maxima, indices = flow[2].get_maxima()
  ```

  Just to check that everything works properly, we can calculate covariance between the generated sources and the output (should be approximately 1):

  ```
  >>> out = flow.execute(x)
  >>> cov = mdp.numx.amax(abs(mdp.utils.cov2(inp[:,:5], out)))
  >>> print cov
  [ 0.98992083  0.99244511  0.99227319  0.99663185  0.9871812 ]
  ```

  The `HitParadeNode` is an analysis node and as such does not interfere with the data flow.

Flow inversion

Flows can be inverted by calling their `inverse` method. In the case where the flow contains non-invertible nodes, trying to invert it would raise an exception. In this case, however, all nodes are invertible. We can reconstruct the mix by inverting the flow:

```
>>> rec = flow.inverse(out)
```

Calculate covariance between input mix and reconstructed mix: (should be approximately 1)

```
>>> cov = mdp.numx.amax(abs(mdp.utils.cov2(x/mdp.numx.std(x,0),
...                                        rec/mdp.numx.std(rec,0))))
>>> print cov
[ 0.99839606  0.99744461  0.99616208  0.99772863  0.99690947
  0.99864056  0.99734378  0.98722502  0.98118101  0.99407939
  0.99683096  0.99756988  0.99664384  0.99723419  0.9985529
  0.99829763  0.9982712   0.99721741  0.99682906  0.98858858]
```

Flows are container type objects

Flows are Python container type objects, very much like lists, i.e., you can loop through them:

```
>>> for node in flow:
...     print repr(node)
...
PCANode(input_dim=20, output_dim=5, dtype='float64')
CuBICANode(input_dim=5, output_dim=5, dtype='float64')
HitParadeNode(input_dim=5, output_dim=5, dtype='float64')
>>>
```

You can get slices, `pop`, `insert`, and `append` nodes like you would do with lists:

```
>>> len(flow)
3
>>> print flow[::2]
[PCANode, HitParadeNode]
>>> nodetoberemoved = flow.pop(-1)
>>> nodetoberemoved
HitParadeNode(input_dim=5, output_dim=5, dtype='float64')
>>> len(flow)
2
```

Finally, you can concatenate flows:

```
>>> dummyflow = flow[1:].copy()
>>> longflow = flow + dummyflow
>>> len(longflow)
3
```

The returned flow must always be consistent, i.e. input and output dimensions of successive nodes always have to match. If you try to create an inconsistent flow you'll get an error.

Crash recovery

If a node in a flow fails, you'll get a traceback that tells you which node has failed. You can also switch the crash recovery capability on. If something goes wrong you'll end up with a pickle dump of the flow, that can be later inspected.

To see how it works let's define a bogus node that always throws an `Exception` and put it into a flow:

```
>>> class BogusExceptNode(mdp.Node):
...     def train(self,x):
...         self.bogus_attr = 1
...         raise Exception, "Bogus Exception"
...     def execute(self,x):
...         raise Exception, "Bogus Exception"
...
>>> flow = mdp.Flow([BogusExceptNode()])
```

Switch on crash recovery:

```
>>> flow.set_crash_recovery(1)
```

Attempt to train the flow:

```
>>> flow.train([None])
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  [...]
mdp.linear_flows.FlowExceptionCR:
----------------------------------------
! Exception in node #0 (BogusExceptNode):
Node Traceback:
Traceback (most recent call last):
  [...]
Exception: Bogus Exception
----------------------------------------
A crash dump is available on: "/tmp/MDPcrash_LmISO_.pic"
```

You can give a file name to tell the flow where to save the dump:

```
>>> flow.set_crash_recovery('/home/myself/mydumps/MDPdump.pic')
```

Iterators

Python allows user-defined classes to support iteration, as described in http://docs.python.org/lib/typeiter.html . A convenient implementation of the iterator protocol is provided by generators.

See http://linuxgazette.net/100/pramode.html for an introduction, and http://www.python.org/peps/pep-0255.html for a complete description.

Let us define two bogus node classes to be used as examples of nodes:

```
>>> class BogusNode(mdp.Node):
...     """This node does nothing."""
...     def _train(self, x):
...         pass
...
>>> class BogusNode2(mdp.Node):
...     """This node does nothing. But it's not trainable nor invertible.
...     """
...     def is_trainable(self): return False
...     def is_invertible(self): return False
...
>>>
```

This generator generates `blocks` input blocks to be used as training set. In this example one block is a 2-dimensional time-series. The first variable is [2,4,6,....,1000] and the second one [0,1,3,5,...,999]. All blocks are equal, this of course would not be the case in a real-life example.

In this example we use a progress bar to get progress information.

```
>>> def gen_data(blocks):
...     for i in mdp.utils.progressinfo(xrange(blocks)):
...         block_x = mdp.numx.atleast_2d(mdp.numx.arange(2,1001,2))
...         block_y = mdp.numx.atleast_2d(mdp.numx.arange(1,1001,2))
...         # put variables on columns and observations on rows
...         block = mdp.numx.transpose(mdp.numx.concatenate([block_x,block_y]))
...         yield block
...
>>>
```

Let's define a bogus flow consisting of 2 `BogusNode`:

```
>>> flow = mdp.Flow([BogusNode(),BogusNode()], verbose=1)
```

Train the first node with 5000 blocks and the second node with 3000 blocks. Note that the only allowed argument to `train` is a sequence (list or tuple) of iterators. In case you don't want or need to use incremental learning and want to do a one-shot training, you can use as argument to `train` a single array of data:

block-mode training

```
>>> flow.train([gen_data(5000),gen_data(3000)])
Training node #0 (IdentityNode)
[==================================100%==================================>]

Training finished
Training node #1 (IdentityNode)
[==================================100%==================================>]

Training finished
Close the training phase of the last node
```

one-shot training using one single set of data for both nodes

```
>>> flow = mdp.Flow([BogusNode(),BogusNode()])
>>> block_x = mdp.numx.atleast_2d(mdp.numx.arange(2,1001,2))
>>> block_y = mdp.numx.atleast_2d(mdp.numx.arange(1,1001,2))
>>> sin-
gle_block = mdp.numx.transpose(mdp.numx.concatenate([block_x,block_y]))
>>> flow.train(single_block)
```

If your flow contains non-trainable nodes, you must specify a `None` iterator for the non-trainable nodes:

```
>>> flow = mdp.Flow([BogusNode2(),BogusNode()], verbose=1)
>>> flow.train([None, gen_data(5000)])
Training node #0 (BogusNode2)
Training finished
Training node #1 (IdentityNode)
[==================================100%==================================>]

Training finished
Close the training phase of the last node
```

If in this case you try the one-shot training you'll get a warning like the following one:

```
>>> flow = mdp.Flow([BogusNode2(),BogusNode()], verbose=1)
>>> flow.train(single_block)
Training node #0 (BogusNode2)
/.../linear_flows.py:94: MDPWarning:
! Node 0 in not trainable
You probably need a 'None' generator for this node. Continuing anyway.
  warnings.warn(wrnstr, mdp.MDPWarning)
Training finished
Training node #1 (IdentityNode)
Training finished
Close the training phase of the last node
```

You can get rid of this warning either by doing what the warning asks you, namely use the iterator syntax and provide a `None` iterator for the non-trainable nodes, or by switching off MDP warnings altogether:

```
>>> import warnings
>>> warnings.filterwarnings("ignore",'.*',mdp.MDPWarning)
>>> flow = mdp.Flow([BogusNode2(),BogusNode()], verbose=1)
>>> flow.train(single_block)
Training node #0 (BogusNode2)
Training finished
Training node #1 (IdentityNode)
Training finished
Close the training phase of the last node
```

To switch on `MDPWarnings` again:

```
>>> warnings.filterwarnings("always",'.*',mdp.MDPWarning)
```

Iterators can be used also for execution (and inversion):

```
>>> flow = mdp.Flow([BogusNode(),BogusNode()], verbose=1)
>>> flow.train([gen_data(1), gen_data(1)])
Training node #0 (BogusNode2)
Training finished
Training node #1 (IdentityNode)
[==================================100%==================================>]

Training finished
Close the training phase of the last node
>>> output = flow.execute(gen_data(1000))
[==================================100%==================================>]
>>> output = flow.inverse(gen_data(1000))
[==================================100%==================================>]
```

Execution and inversion can be done in one-shot mode also. Note that since training is finished you are not going to get a warning

```
>>> output = flow.execute(single_block)
>>> output = flow.inverse(single_block)
```

If a node requires multiple training phases (e.g., `GaussianClassifierNode`), `Flow` automatically takes care of reusing the iterator multiple times. In this case generators are not allowed, since they

"expire" after yielding the last data block. If you try to restart them, they raise a `StopIteration` exception. General iterators, instead, can always be restarted. For example, you can loop over a list as many times as you need.

However, it is fairly easy to wrap a generator in a simple iterator if you need to:

```
>>> class SimpleIterator(object):
...     def __init__(self, blocks):
...         self.blocks = blocks
...     def __iter__(self):
...         # this is a generator
...         for i in range(self.blocks):
...             yield generate_some_data()
>>>
```

Note that if you use random numbers within the iterator, you usually would like to reset the random number generator to produce the same sequence every time:

```
>>> class RandomIterator(object):
...     def __init__(self):
...         self.state = None
...     def __iter__(self):
...         if self.state is None:
...             self.state = mdp.numx_rand.get_state()
...         else:
...             mdp.numx_rand.set_state(self.state)
...         for i in range(2):
...             yield mdp.numx_rand.random((1,4))
>>> iterator = RandomIterator()
>>> for x in iterator: print x
...
[[ 0.99586495  0.53463386  0.6306412   0.09679571]]
[[ 0.51117469  0.46647448  0.95089738  0.94837122]]
>>> for x in iterator: print x
...
[[ 0.99586495  0.53463386  0.6306412   0.09679571]]
[[ 0.51117469  0.46647448  0.95089738  0.94837122]]
```

Checkpoints

It can sometimes be useful to execute arbitrary functions at the end of the training or execution phase, for example to save the internal structures of a node for later analysis. This can easily be done by defining a `CheckpointFlow`. As an example imagine the following situation: you want to perform Principal Component Analysis (PCA) on your data to reduce the dimensionality. After this you want to expand the signals into a nonlinear space and then perform Slow Feature Analysis to extract slowly varying signals. As the expansion will increase the number of components, you don't want to run out of memory, but at the same time you want to keep as much information as possible after the dimensionality reduction. You could do that by specifying the percentage of the total input variance that has to be conserved in the dimensionality reduction. As the number of output components of the PCA node now can become as large as the that of the input components, you want to check, after training the PCA node, that this number is below a certain threshold. If this is not the case you want to abort the execution and maybe start again requesting less variance to be kept.

Let start defining a generator to be used through the whole example:

```
>>> def gen_data(blocks,dims):
```

```
...     mat = mdp.numx_rand.random((dims,dims))-0.5
...     for i in xrange(blocks):
...         # put variables on columns and observations on rows
...         block = mdp.utils.mult(mdp.numx_rand.random((1000,dims)), mat)
...         yield block
...
>>>
```

Define a `PCANode` which reduces dimensionality of the input, a `PolynomialExpansionNode` to expand the signals in the space of polynomials of degree 2 and a `SFANode` to perform SFA:

```
>>> pca = mdp.nodes.PCANode(output_dim=0.9)
>>> exp = mdp.nodes.PolynomialExpansionNode(2)
>>> sfa = mdp.nodes.SFANode()
```

As you see we have set the output dimension of the `PCANode` to be `0.9`. This means that we want to keep at least 90% of the variance of the original signal. We define a `PCADimensionExceededException` that has to be thrown when the number of output components exceeds a certain threshold:

```
>>> class PCADimensionExceededException(Exception):
...     """Exception base class for PCA exceeded dimensions case."""
...     pass
...
>>>
```

Then, write a `CheckpointFunction` that checks the number of output dimensions of the `PCANode` and aborts if this number is larger than `max_dim`:

```
>>> class CheckPCA(mdp.CheckpointFunction):
...     def __init__(self,max_dim):
...         self.max_dim = max_dim
...     def __call__(self,node):
...         node.stop_training()
...         act_dim = node.get_output_dim()
...         if act_dim > self.max_dim:
...             errstr = 'PCA output dimensions exceeded maximum '+\
...                      '(%d > %d)'%(act_dim,self.max_dim)
...             raise PCADimensionExceededException, errstr
...         else:
...             print 'PCA output dimensions = %d'%(act_dim)
...
>>>
```

Define the CheckpointFlow:

```
>>> flow = mdp.CheckpointFlow([pca, exp, sfa])
```

To train it we have to supply 3 generators and 3 checkpoint functions:

```
>>> flow.train([gen_data(10, 50), None, gen_data(10, 50)],
...            [CheckPCA(10), None, None])
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
  [...]
__main__.PCADimensionExceededException: PCA output dimensions exceeded maxi-
mum (25 > 10)
```

The training fails with a `PCADimensionExceededException`. If we only had 12 input dimensions instead of 50 we would have passed the checkpoint:

```
>>> flow[0] = mdp.nodes.PCANode(output_dim=0.9)
>>> flow.train([gen_data(10, 12), None, gen_data(10, 12)],
...            [CheckPCA(10), None, None])
PCA output dimensions = 6
```

We could use the built-in `CheckpoinSaveFunction` to save the `SFANode` and analyze the results later :

```
>>> pca = mdp.nodes.PCANode(output_dim=0.9)
>>> exp = mdp.nodes.PolynomialExpansionNode(2)
>>> sfa = mdp.nodes.SFANode()
>>> flow = mdp.CheckpointFlow([pca, exp, sfa])
>>> flow.train([gen_data(10, 12), None, gen_data(10, 12)],
...            [CheckPCA(10),
...             None,
...             mdp.CheckpointSaveFunction('dummy.pic',
...                                        stop_training = 1,
...                                        protocol = 0)])
...
PCA output dimensions = 7
```

We can now reload and analyze the `SFANode`:

```
>>> fl = file('dummy.pic')
>>> import cPickle
>>> sfa_reloaded = cPickle.load(fl)
>>> sfa_reloaded
SFANode(input_dim=35, output_dim=35, dtype='d')
```

Don't forget to clean the rubbish:

```
>>> fl.close()
>>> import os
>>> os.remove('dummy.pic')
```

A real life example (Logistic maps)

We show an application of Slow Feature Analysis to the analysis of non-stationary time series. We consider a chaotic time series generated by the logistic map based on the logistic equation (a demographic model of the population biomass of species in the presence of limiting factors such as food supply or disease), and extract the slowly varying parameter that is hidden behind the time series. This example reproduces some of the results reported in: Laurenz Wiskott, *Estimating Driving Forces of Nonstationary Time Series with Slow Feature Analysis*. arXiv.org e-Print archive, http://arxiv.org/abs/cond-mat/0312317

Generate the slowly varying driving force, a combination of three sine waves (freqs: 5, 11, 13 Hz), and define a function to generate the logistic map

```
>>> p2 = mdp.numx.pi*2
>>> t = mdp.numx.linspace(0,1,10000,endpoint=0) # time axis 1s, sampler-
ate 10KHz
>>> dforce = mdp.numx.sin(p2*5*t) + mdp.numx.sin(p2*11*t) + mdp.numx.sin(p2*13*t)
>>> def logistic_map(x,r):
```

```
...     return r*x*(1-x)
...
>>>
```

Note that we define `series` to be a two-dimensional array. Inputs to MDP must be two-dimensional arrays with variables on columns and observations on rows. In this case we have only one variable:

```
>>> series = mdp.numx.zeros((10000,1),'d')
```

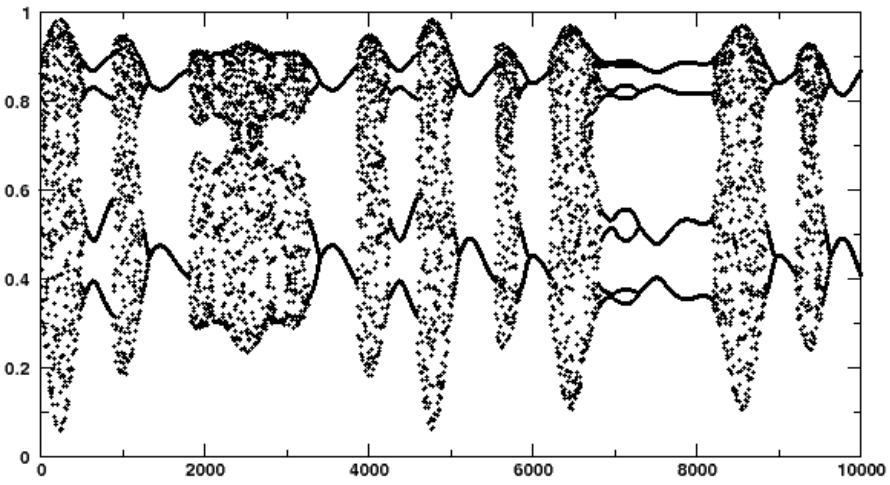
Fix the initial condition:

```
>>> series[0] = 0.6
```

Generate the time-series using the logistic equation the driving force modifies the logistic equation parameter `r`:

```
>>> for i in range(1,10000):
...     series[i] = logistic_map(series[i-1],3.6+0.13*dforce[i])
...
>>>
```

If you have a plotting package `series` should look like this:



Define a flow to perform SFA in the space of polynomials of degree 3. We need a node that embeds the time-series in a 10 dimensional space, where different variables correspond to time-delayed copies of the original time-series: the `TimeFramesNode(10)`. Then we need a node that expands the new signal in the space of polynomials of degree 3: the `PolynomialExpansionNode(3)`. Finally we perform SFA onto the expanded signal and keep the slowest feature: `SFANode(output_dim=1)`. We also measure the *slowness* of the input time-series and of the slow feature obtained by SFA. Therefore we put at the beginning and at the end of the sequence an *analysis node* that computes the *eta-value* (a measure of slowness) of its input (see docs for the definition of eta-value): the `EtaComputerNode()`:

```
>>> sequence = [mdp.nodes.EtaComputerNode(),
...              mdp.nodes.TimeFramesNode(10),
...              mdp.nodes.PolynomialExpansionNode(3),
...              mdp.nodes.SFANode(output_dim=1),
...              mdp.nodes.EtaComputerNode()]
```

```
...
>>>
>>> flow = mdp.Flow(sequence, verbose=1)
```

Since the time-series is short enough to be kept in memory we don't need to define generators and we can feed the flow directly with the whole signal:

```
>>> flow.train(series)
```

Since the second and the third nodes are not trainable we are going to get two warnings (`Training Interrupted`). We can safely ignore them. Execute the flow to get the slow feature

```
>>> slow = flow.execute(series)
```

The slow feautre should match the driving force up to a scaling factor, a constant offset and the sign. To allow a comparison we rescale the driving force to have zero mean and unit variance:

```
>>> resc_dforce = (dforce - mdp.numx.mean(dforce,0))/mdp.numx.std(dforce,0)
```

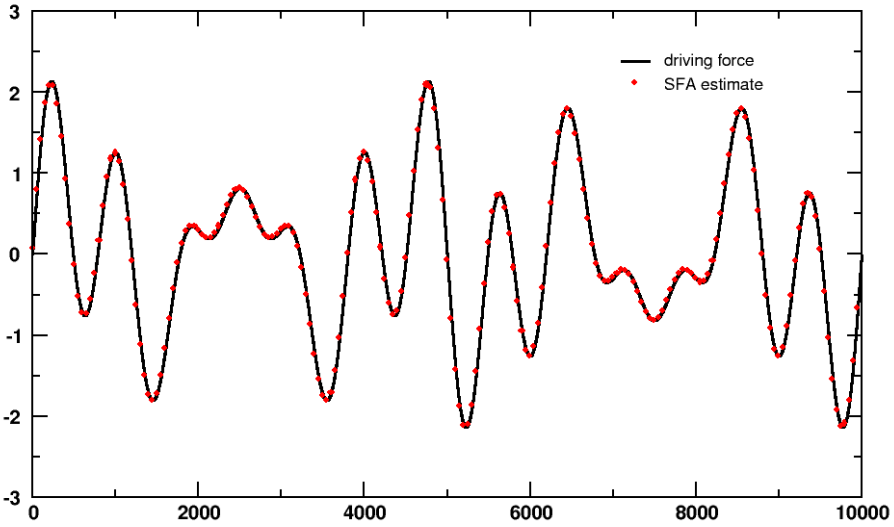
Print covariance between the rescaled driving force and the slow feature. Note that embedding the time-series with 10 time frames leads to a time-series with 9 observations less:

```
>>> mdp.utils.cov2(resc_dforce[:-9],slow)
0.99992501533859179
```

Print the *eta-values* of the chaotic time-series and of the slow feature

```
>>> print 'Eta value (time-series): ', flow[0].get_eta(t=10000)
Eta value (time-series):  [ 3002.53380245]
>>> print 'Eta value (slow feature): ', flow[-1].get_eta(t=9996)
Eta value (slow feature):  [ 10.2185087]
```

If you have a plotting package you could plot `resc_dforce` together with `slow` and see that they match perfectly:



Another real life example (Growing neural gas)

We generate uniformly distributed random data points confined on different 2-D geometrical objects. The Growing Neural Gas Node builds a graph with the same topological structure.

Fix the random seed to obtain reproducible results:

```
>>> mdp.numx_rand.seed(1266090063)
```

Some functions to generate uniform probability distributions on different geometrical objects:

```
>>> def uniform(min_, max_, dims):
...     """Return a random number between min_ and max_ ."""
...     return mdp.numx_rand.random(dims)*(max_-min_)+min_
...
>>> def circumference_distr(center, radius, n):
...     """Return n random points uniformly distributed on a circumfer-
ence."""
...     phi = uniform(0, 2*mdp.numx.pi, (n,1))
...     x = radius*mdp.numx.cos(phi)+center[0]
...     y = radius*mdp.numx.sin(phi)+center[1]
...     return mdp.numx.concatenate((x,y), axis=1)
...
>>> def circle_distr(center, radius, n):
...     """Return n random points uniformly distributed on a circle."""
...     phi = uniform(0, 2*mdp.numx.pi, (n,1))
...     sqrt_r = mdp.numx.sqrt(uniform(0, radius*radius, (n,1)))
...     x = sqrt_r*mdp.numx.cos(phi)+center[0]
...     y = sqrt_r*mdp.numx.sin(phi)+center[1]
...     return mdp.numx.concatenate((x,y), axis=1)
...
>>> def rectangle_distr(center, w, h, n):
...     """Return n random points uniformly distributed on a rectangle."""
...     x = uniform(-w/2., w/2., (n,1))+center[0]
...     y = uniform(-h/2., h/2., (n,1))+center[1]
...     return mdp.numx.concatenate((x,y), axis=1)
...
>>> N = 2000
```

Explicitly collect random points from some distributions:

- Circumferences:

```
>>> cf1 = circumference_distr([6,-0.5], 2, N)
>>> cf2 = circumference_distr([3,-2], 0.3, N)
```

- Circles:

```
>>> cl1 = circle_distr([-5,3], 0.5, N/2)
>>> cl2 = circle_distr([3.5,2.5], 0.7, N)
```

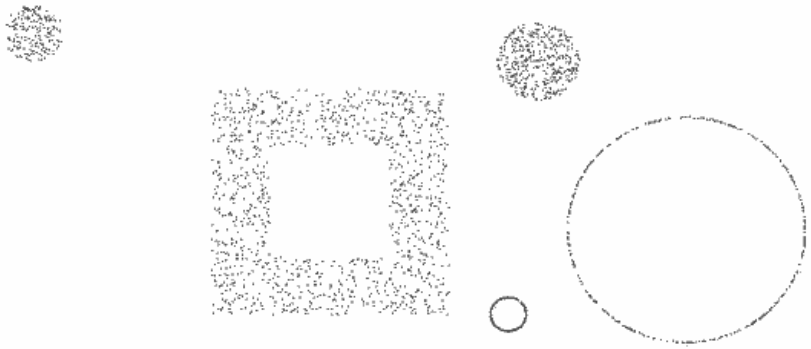
- Rectangles:

```
>>> r1 = rectangle_distr([-1.5,0], 1, 4, N)
>>> r2 = rectangle_distr([+1.5,0], 1, 4, N)
>>> r3 = rectangle_distr([0,+1.5], 2, 1, N/2)
>>> r4 = rectangle_distr([0,-1.5], 2, 1, N/2)
```

Shuffle the points to make the statistics stationary

```
>>> x = mdp.numx.concatenate([cf1, cf2, cl1, cl2, r1,r2,r3,r4], axis=0)
>>> x = mdp.numx.take(x,mdp.numx_rand.permutation(x.shape[0]))
```

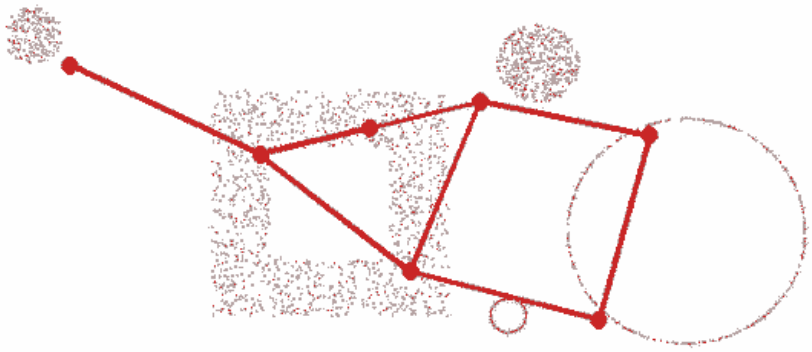
If you have a plotting package `x` should look like this:



Create a `GrowingNeuralGasNode` and train it:

```
>>> gng = mdp.nodes.GrowingNeuralGasNode(max_nodes=75)
```

The initial distribution of nodes is randomly chosen:

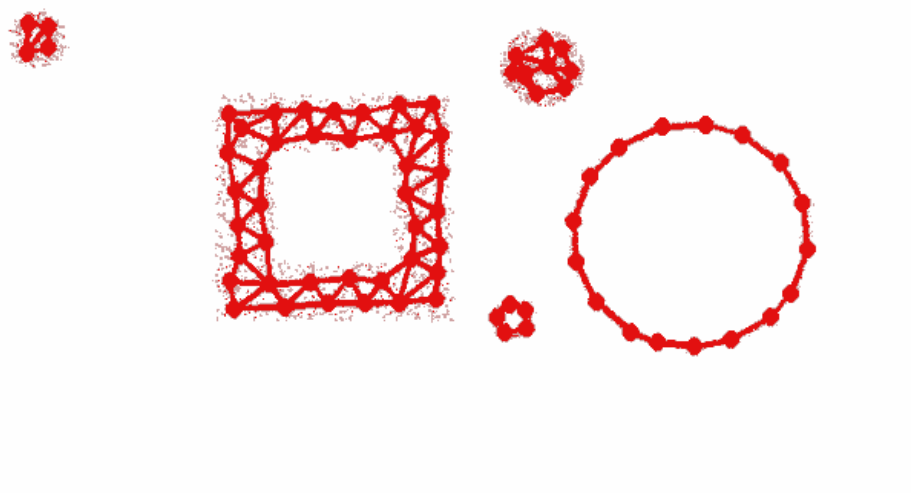


The training is performed in small chunks in order to visualize the evolution of the graph:

```
>>> STEP = 500
>>> for i in range(0,x.shape[0],STEP):
...     gng.train(x[i:i+STEP])
...     # [...] plotting instructions
...
>>> gng.stop_training()
```

See here the animation of training.

Visualizing the neural gas network, we'll see that it is adapted to the topological structure of the data distribution:



Calculate the number of connected components:

```
>>> n_obj = len(gng.graph.connected_components())
>>> print n_obj
5
```

utils section

To Do

In this last section we want to give you an overview about our plans for the development of MDP:

- Add more data processing algorithms.
- Extend the linear flows to handle general acyclic graphs of nodes.
- Actual use of the graph structure will be possible only in presence of an easy and intuitive GUI :)
- Wait for a good guy who wants to contribute a `CovarianceMatrix` class that uses some of the fancy sum algorithms to avoid round off errors when adding many numbers.

A final remark

If you want to contribute some code or a new algorithm, please do not hesitate to submit it!

Generated on: 2006-06-28 18:04 UTC. Generated by Docutils from reStructuredText source.