

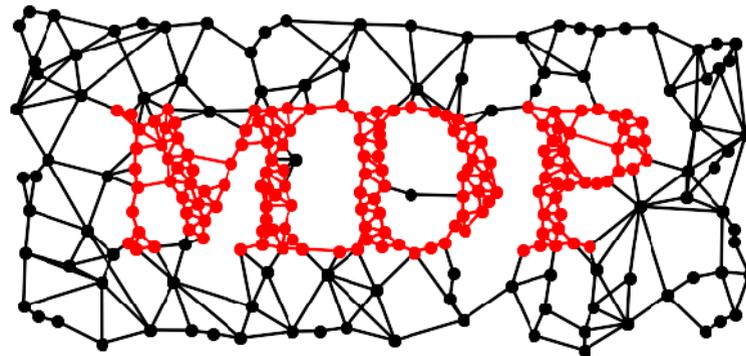
MDP: Modular toolkit for Data Processing (and its new features)

Pietro Berkes, Rike-Benjamin Schuppner,
Niko Wilbert, Tiziano Zito



Background

- Library of widely used data processing algorithms, and a framework to combine them according to a pipeline analogy.
- origin: Computational Neuroscience and Machine Learning, supported by Prof. Laurenz Wiskott (HU Berlin, now Bochum)
- documentation and unittest coverage, tutorial on homepage
- first release 2004, 15k+ downloads, available in Debian, MacPorts and Python(x,y)



Talk Overview

- Introducing the basic building blocks:

- ◆ Nodes and Flows
- ◆ Hierarchical Networks
- ◆ Parallelization

- New features:

- ◆ Node Extensions
- ◆ bidirectional data flow with BiMDP

Building blocks: Node

Node represents data processing algorithm, interface methods:

train (optional)

support for multiple phases, chunks, supervised training

execute

map n dimensional input to m dimensional output

inverse (optional)

inverse of execute method

data format: 2d numpy arrays

(1st index for samples, 2nd index for channels)

Nodes do automatic checks and conversions (dimensions, dtype).

Building blocks: Node

Example:

Principal Component Analysis (PCA)

reduce dimension of data from 10 to 5:

```
>>> import mdp
>>> import numpy as np
>>> data = np.random.random((50,10)) # 50 data points
>>> node = mdp.nodes.PCANode(output_dim=5)
>>> node.train(data)
>>> proj_data = node.execute(data)
```

Building blocks: Node

Some available nodes:

PCA (standard, NIPALS)
ICA (FastICA, CuBICA, JADE, TDSEP)
Locally Linear Embedding
Hessian Locally Linear Embedding
Fisher Discriminant Analysis
Slow Feature Analysis
Independent Slow Feature Analysis
Restricted Boltzmann Machine
Growing Neural Gas
Factor Analysis
Gaussian Classifiers
Polynomial Expansion
Time Frames
Hit Parades
Noise
...

Or write your own node (and contribute it :-).

Building blocks: Node

Write your own node class:

```
class MyNode(Node):  
  
    def _train(self, x):  
        # training code  
  
    def _execute(self, x):  
        # execution code
```

Node base class takes care of the rest.

Building blocks: Flow

Combine nodes in a **Flow**:

```
>>> flow = PCANode() + SFANode() + FastICANode()  
>>> flow.train(train_data)  
>>> test_result = flow.execute(test_data)
```

- management of training, execution, inversion
- arrays or iterables for input

Building blocks: Network

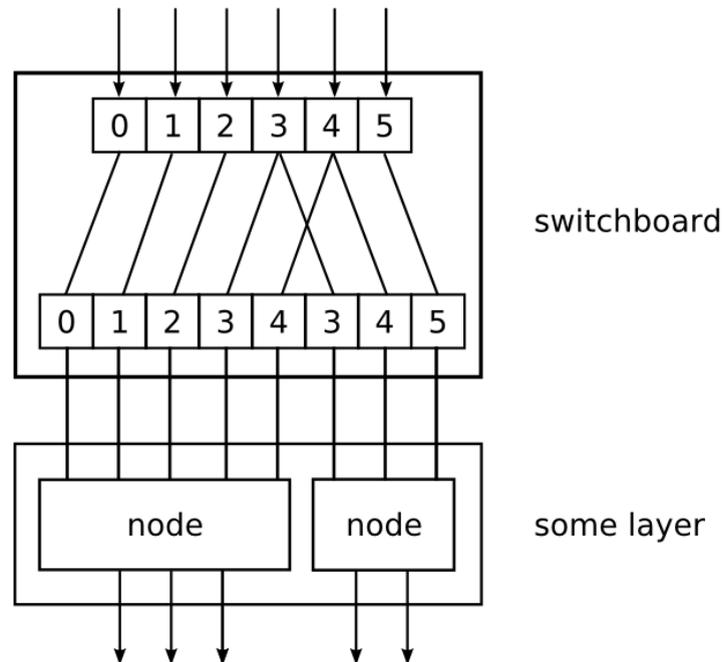
mdp.hinet package for hierarchical networks

Layer (combine nodes horizontally, in parallel)

Switchboard (routing between layers)

FlowNode (combine nodes into a “supernode”)

All these classes are nodes, combine them as you want.

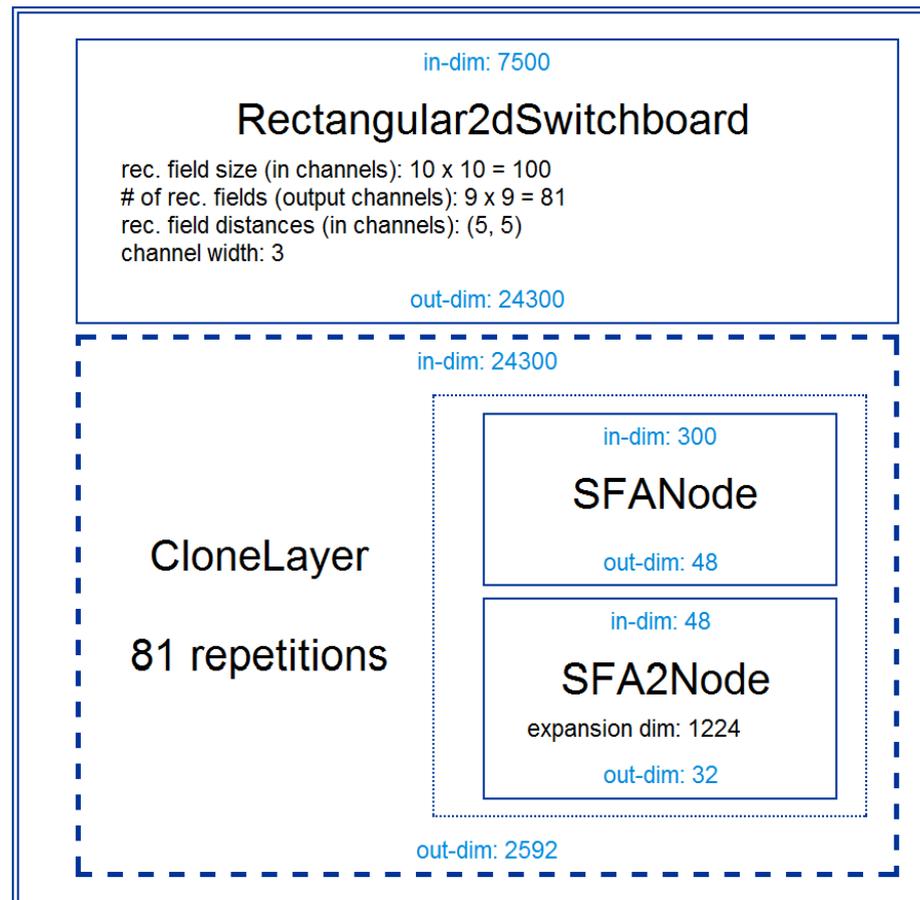


Building blocks: Network

HTML representation of your network:

```
>>> mdp.hinet.show_flow(flow)
```

Use this in your reports or GUI.



Parallelization

- training: nodes provide `_fork` and `_join` methods, `fork` → parallel training → `join` (“embarrassingly parallel” problem)
- execution: node copies are used
- use multiple cores or multiple machines (e.g. via Parallel Python library)
- abstract scheduler API (easy to write adaptor)

Example:

```
>>> nodes = [PCANode(output_dim=10), SFANode()]
>>> flow = mdp.parallel.ParallelFlow(nodes)
>>> scheduler = mdp.parallel.ProcessScheduler() # one process per core
>>> flow.train(data, scheduler)
```

Real World Example

- object recognition system, working on 155x155 pixel image sequences
- hierarchical network with nested nodes
- several GB of training data for each layer
- training is distributed over network, takes multiple hours



[Franzius, M., Wilbert, N., and Wiskott, L., 2008]

New feature: Node Extensions

Problem: How to add new aspects to nodes?

- Parallelization: add `_fork` and `_join` methods to node classes
- HTML representation: add `_html_representation` method

Solution: Node Extension Mechanism

inside `ParallelFlow`:

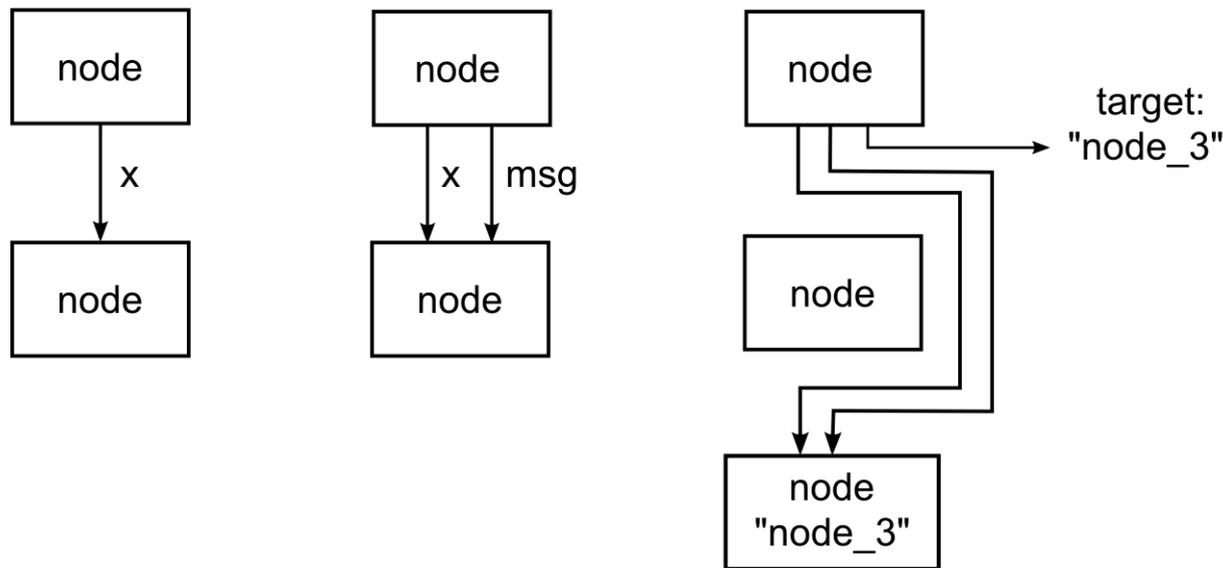
```
>>> mdp.activate_extension("parallel")
>>> # do parallel stuff with the new fork and join methods
>>> mdp.deactivate_extension("parallel")
```

Dynamically add class attributes (methods) to the supported node classes.

Define your own custom extensions

New feature: BiMDP

- transport additional information in `msg` dictionary,
- nodes can specify a target node, enabling error backpropagation and loops
- helpful functionality to get the data to the right node
- compatible with `mpd.parallel` and `mdp.hinet` package



New feature: BiMDP

HTML+JS based inspector for debugging and analysing

The screenshot displays the 'Execution Inspection' tool interface. At the top, there is a browser window with the address bar showing 'file:///C:/Users/Niko/appdata/local/temp/MDP_oo4sja/execution_inspection.html'. Below the browser window, the tool title 'Execution Inspection' is centered. A control panel includes a dropdown menu for the current file ('execution_inspection_0.html'), navigation buttons ('<<', '<', 'Start', '>', '>>'), and a delay input field set to '500 ms'. The main content area is divided into two sections: 'flow state' and 'execute arguments'. The 'flow state' section shows a hierarchical diagram of nodes: a top-level 'Rectangular2dSwitchboard' node (in-dim: 10000, out-dim: 32400) containing a 'CloneLayer' node (in-dim: 32400, out-dim: 810) which in turn contains '81 repetitions' of a 'NormalNoiseNode' (in-dim: 400, out-dim: 400) and an 'SFANode' (in-dim: 400, out-dim: 10). The 'execute arguments' section displays a list of numerical values for 'x' with a shape of (10, 10000). The 'execute result' section shows a similar list of numerical values for 'x' with a shape of (10, 32400).

works in all browsers (Chrome 5.0 currently with restriction)

Thank you!

Upcoming: MDP Sprint

Join us for a coding sprint in Berlin.

19th - 23th July 2010

More information on the MDP homepage (or ask us).



New feature: Node Extensions

How to define extension methods for a Node class?

- Use multiple inheritance:

```
class ParallelPCANode(ParallelExtensionNode, mdp.nodes.PCANode):  
  
    def _join(self, forked_node):  
        # join code goes here
```

Metaclass magic to register which methods are available.

- Use the function decorator for single methods:

```
@mdp.extension_method("parallel", mdp.nodes.PCANode)  
def _join(self, forked_node):  
    # join code goes here
```