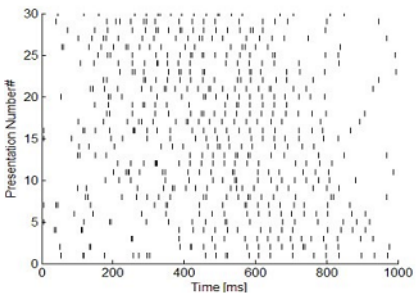
Python in Computational Neuroscience & Modular toolkit for Data Processing (MDP)

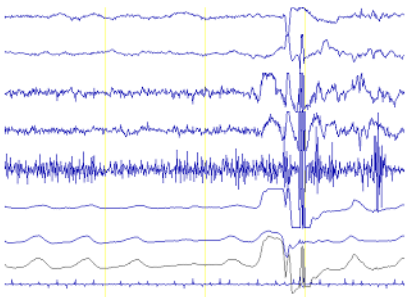
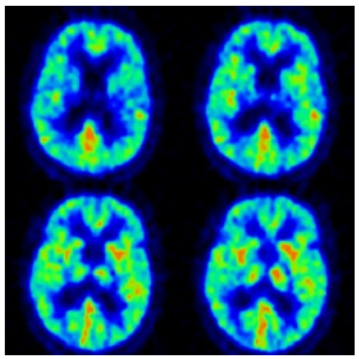
Niko Wilbert (HU Berlin)

Python in Computational Neuroscience

Computational Neuroscience

Computational Neuroscience ist ein wachsender Bereich der life sciences.
Wird in Deutschland gefördert durch das BMBF im Rahmen des Bernstein Netzwerkes.
"Computational" bezieht sich die Erforschung der Informationsverarbeitung im Gehirn.




Unsere Gruppe versucht die Funktionen des Visuellen Systems zu modellieren, mit mathematischen Ansätze und Computermodellen.
Verbindung zu machine learning

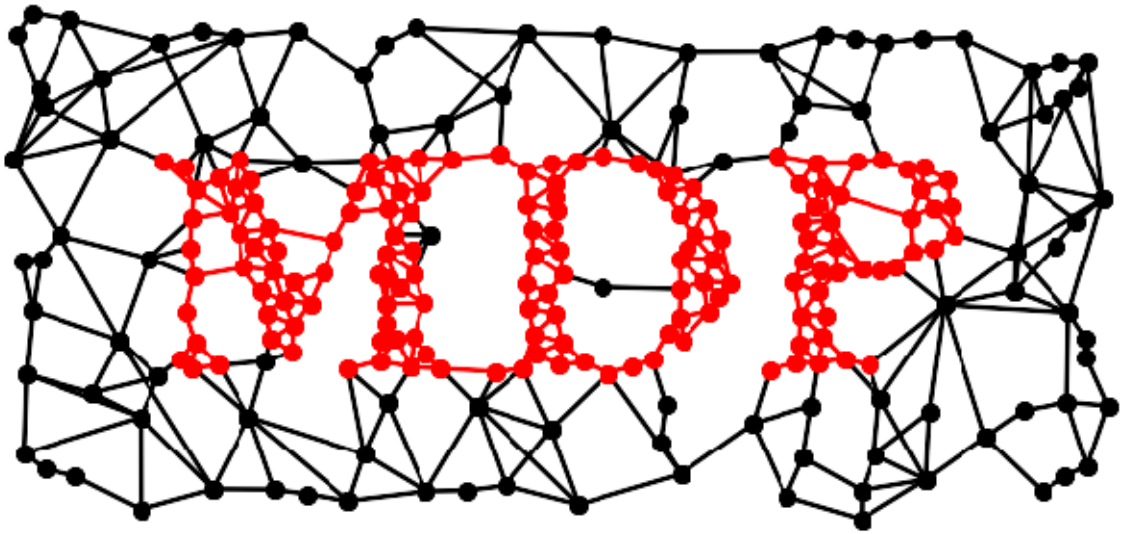
Python in Computational Neuroscience

Wissenschaftler in diesem Bereich benötigen Computer für die Datenanalyse oder Simulationen,
von einfachen Matlab Skripten bis zur Nutzung von Supercomputern (z.B. IBM Blue Gene/L in Lausanne).

Python wird zunehmend populär:

- Ersetzt Matlab in der Datenanalyse und einfachen Simulationen.
- Produktives Frontend für komplexe Simulatoren (z.B., PyNN ist ein Interface zu Standard-Simulationssoftware wie NEURON).

Modular toolkit for Data Processing



Hintergrund

- Open Source library (LGPL)
- erste Veröffentlichung 2004 durch Pietro Berkes und Tiziano Zito
- 15k+ Downloads, verfügbar in Debian, Ubuntu, MacPorts, Python(x,y)
- Entstanden in und unterstützt durch die Forschungsgruppe von Prof. Wiskott, auch mit dem Ziel Forschungsarbeiten reproduzierbar zu machen und Zusammenarbeit zu erleichtern.
- Anwendungen auch außerhalb der Neurowissenschaften.

Überblick

1. Einführung der Grundbausteine von MDP
2. Parallelisierung
3. Beispielanwendung

Grundbaustein: Node

Node: Grundbaustein der Datenverarbeitung,
die Node Klassen entsprechen Algorithms, public API:

train (optional)
: Unterstützt mehrere Trainingsphasen, batch, online, chunks,
überwacht, unüberwacht

execute
: bildet n dimensionalen input auf m dimensionalen output ab

inverse (optional)
: inverse Abbildung zu execute

Datenformat: 2d numpy arrays
(Erster index für samples, zweiter Index für Kanäle)
Automatische Tests und Unformungen (dimensions, dtype).

Node Beispiel

Beispiel: Principal Component Analysis (PCA)
reduziere Daten Dimensionalität von 10 auf 5:

```
>>> import mdp
>>> import numpy as np
>>> data = np.random.random((50,10))  # 50 data points
>>> node = mdp.nodes.PCANode(output_dim=5,
...                          dtype='float32')
>>> node.train(data)
>>> proj_data = node.execute(data)
```

Abkürzung:

```
>>> import mdp
>>> import numpy as np
>>> data = np.random.random((50,10))  # 50 data points
>>> proj_data = mdp.pca(data, output_dim=5, dtype='float32')
```

Nodes in MDP

Einige der verfügbaren 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
...

Oder schreibe eigenen Node (und stelle ihn MDP zur Verfügung :-).

Grundbaustein: Flow

Kombiniere Nodes zu einem **Flow** (data processing pipeline):

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

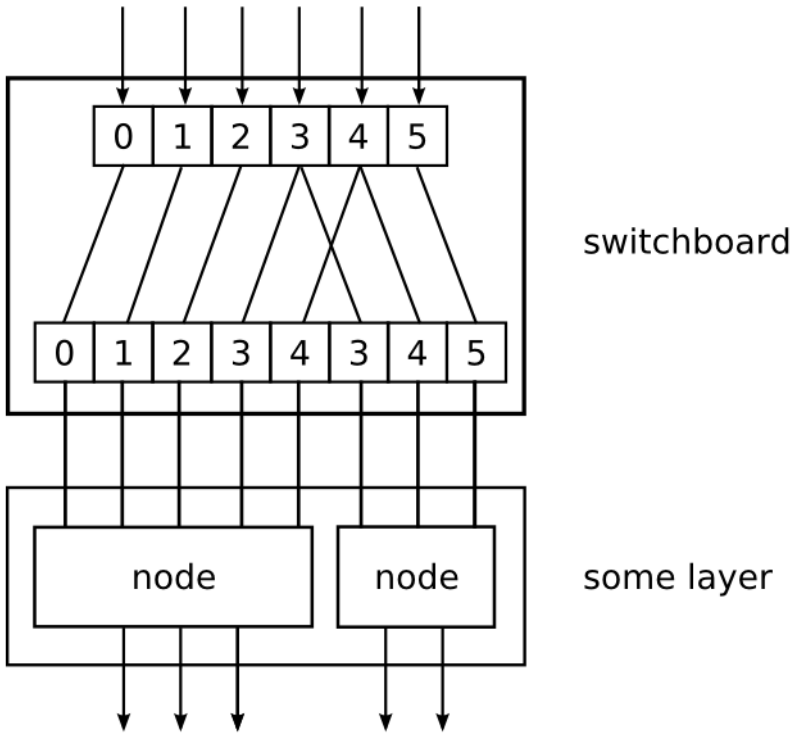
- automatische Verwaltung von Training, Execution, Inversion
- automatische Tests: Dimensionalität und Format
- Benutze arrays oder iterators
- crash recovery, checkpoints

Grundbaustein: Netzwerke

`mdp.hinet` package für hierarchische Netzwerke

Layer (kombiniere Nodes horizontal)
Switchboard (Vernetzung zwischen Layern)
FlowNode (kombiniere Nodes zu einem "supernode")

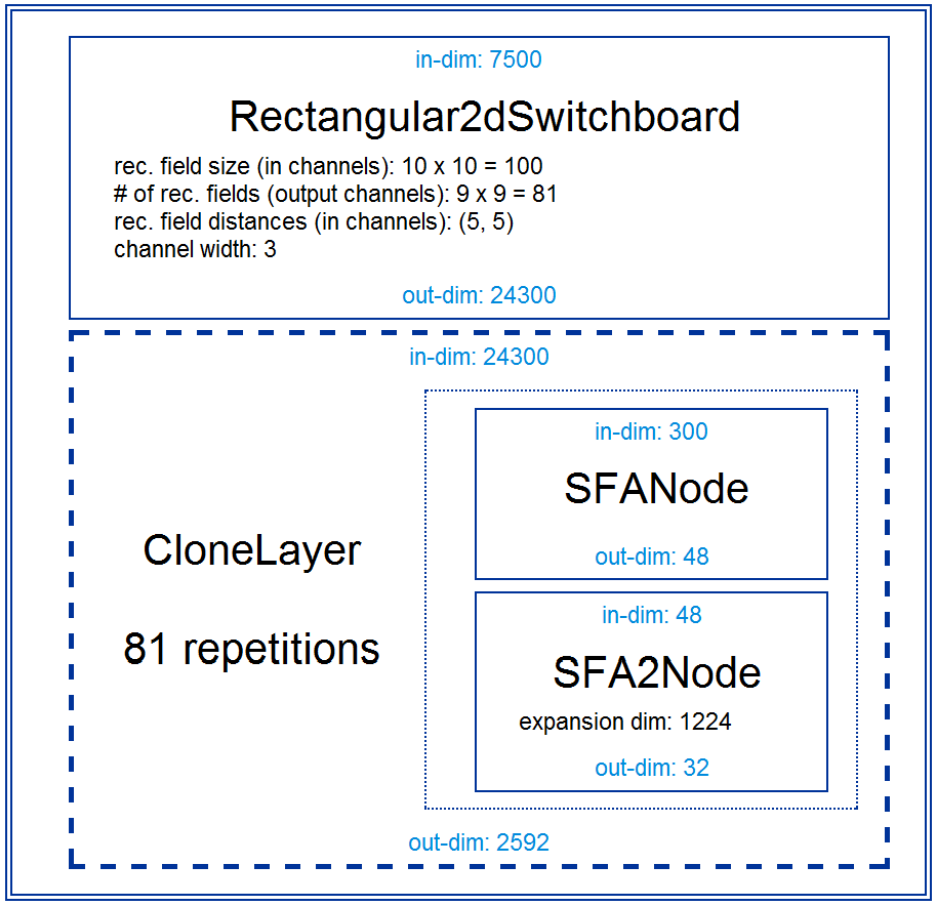
Alle diese Klassen sind Nodes und können beliebig kombiniert werden.



HTML Darstellung eines Netzwerkes

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

Nützlich zum Debuggen, für Dokumentation oder GUI.



Ausblick: BiNet package

Das `mdp.binet` package wird Datenfluß in beide Richtungen erlauben, ermöglicht z.B. error backpropagation und Schleifen.

HTML+JS basierter Inspektor für Debugging und Analysen.

Geplant für MDP 3.0

Parallelisierung

- für “embarrassingly parallel” Probleme:
  Datensätze für das Node Training können unabhängig bearbeitet werden, Resultate werden am Ende kombiniert
  fork Node -> paralleles Training -> join Nodes
- parallel training und execution werden automatisch verwaltet durch einen `ParallelFlow`
  funktioniert für beliebige hierarchische Netzwerke
  Organisation ist wesentlicher Teil der Parallelisierung
- einfache Implementierung für neue Nodes
  (implementiere `_fork` und `_join` Methoden)

Beispiel:

```
>>> pflow = mdp.parallel.ParallelFlow([PCANode(), SFANode()])
>>> scheduler = mdp.parallel.ProcessScheduler(n_processes=4)
>>> pflow.train(data, scheduler)
```

Parallele Schedulers

MDP verwendet eine abstrakte Scheduler API, die Tasks entgegen nimmt und die Ergebnisse bereitstellt (Adapter für externe Scheduler können einfach implementiert werden).

Momentan verfügbare Scheduler:

- Prozess basierte Scheduler für SMP.
- Adapter für Parallel Python library (Prozesse oder Sockets).
- Intern verwenden wir auch einen SSH basierten Scheduler der auf Desktop Rechner im Institut zugreift.

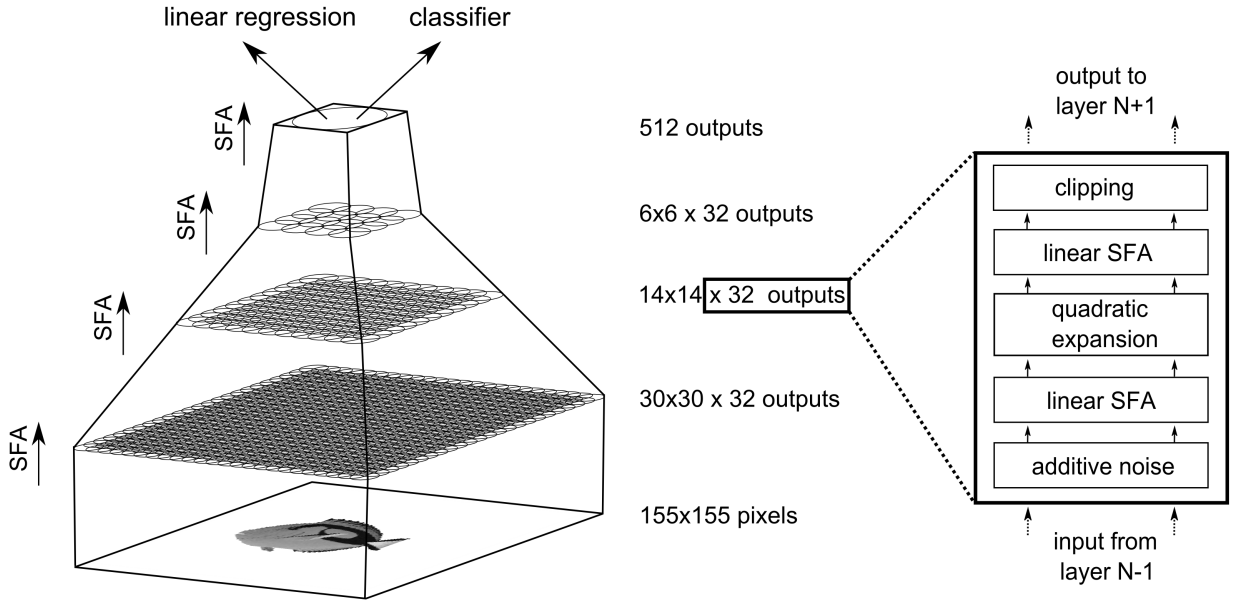
Zukunftsaussichten:

- verwende shared memory mit dem Python multiprocessing Modul
- verwende MPI (via mpi4py)

Und natürlich warten wir auf einfach benutzbare GPU Beschleunigung.

Projektbeispiel

- Modell zur visuellen Objekterkennung,
  trainiert mit 155x155 Pixel großen Bildsequenzen
- mehrere GB an Trainingsdaten pro Trainingsphase.
- Hierarchisches Netzwerk mit geschachtelten Nodes,
  900 FlowNodes in der untersten Schicht
- Training ist parallelisiert, benötigt trotzdem mehrere Stunden

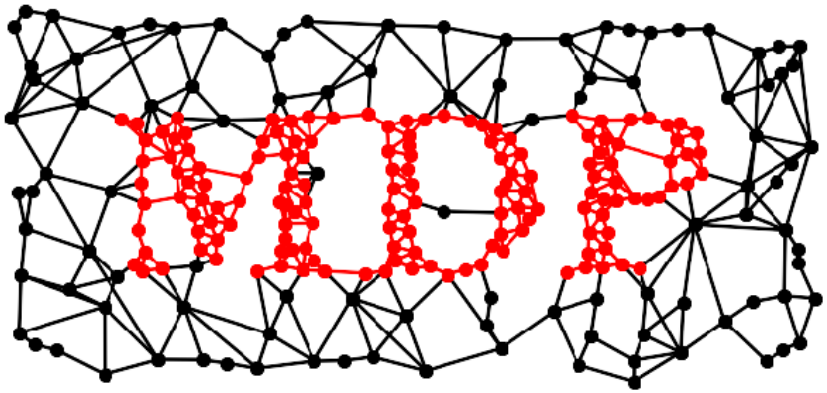


[Franzius, M., Wilbert, N., and Wiskott, L., 2008]

Verwendung von MDP

- ausführliche Dokumentation:
  Tutorial für den Einstieg und fortgeschrittene Aspekte,
  detailierte doc-strings,
  PEP8 compliant, dokumentiert, und pylint-geprüfter Code
- API ist stabil und designed für einfache Integration
- unittest coverage (400+ unit tests)
- minimale dependencies: Python + NumPy
- wird z.B. benutzt in:
  PyMCA (X-ray fluorescence mapping),
  PyMVPA (ML framework for neuroimaging data analysis),
  Chandler (personal organizer application)

Danke!



mdp-toolkit.sourceforge.net