Journal of Computational Science 4 (2013) 345–351

Contents lists available at ScienceDirect

Journal of Computational Science

journal homepage: www.elsevier.com/locate/jocs

Building extensible frameworks for data processing: The case of MDP, Modular toolkit for Data Processing

Niko Wilbert[a,b,*], Tiziano Zito[b,c], Rike-Benjamin Schuppner[b], Zbigniew Jędrzejewski-Szmek[d], Laurenz Wiskott[a,b,e], Pietro Berkes[f]

[a] Institute for Theoretical Biology, Humboldt-Universität zu Berlin, Germany
[b] Bernstein Center for Computational Neuroscience, Berlin, Germany
[c] Modeling of Cognitive Processes, Berlin Institute of Technology, Germany
[d] Institute of Experimental Physics, University of Warsaw, Poland
[e] Institut für Neuroinformatik, Ruhr-Universität Bochum, Germany
[f] National Volen Center for Complex Systems, Brandeis University, Waltham, MA, USA

ARTICLE INFO

Article history:
Received 1 February 2011
Received in revised form 16 October 2011
Accepted 24 October 2011
Available online 30 October 2011

Keywords:
Machine learning
Python
Scientific computing
Computational neuroscience

ABSTRACT

Data processing is a ubiquitous task in scientific research, and much energy is spent on the development of appropriate algorithms. It is thus relatively easy to find software implementations of the most common methods. On the other hand, when building concrete applications, developers are often confronted with several additional chores that need to be carried out beside the individual processing steps. These include for example training and executing a sequence of several algorithms, writing code that can be executed in parallel on several processors, or producing a visual description of the application. The Modular toolkit for Data Processing (MDP) is an open source Python library that provides an implementation of several widespread algorithms and offers a unified framework to combine them to build more complex data processing architectures. In this paper we concentrate on some of the newer features of MDP, focusing on the choices made to automatize repetitive tasks for users and developers. In particular, we describe the support for parallel computing and how this is implemented via a flexible extension mechanism. We also briefly discuss the support for algorithms that require bi-directional data flow.

1. Introduction

Data processing[1] applications often require a combination of several algorithms to achieve the desired result. For example, an application to recognize handwritten digits might require filtering the incoming images, reducing the dimensionality of the data, and classifying the resulting output as one of the digits. Although each step in this chain might be a well-known method and as such likely to be readily available in some form, for the complete application these methods also need to be able to communicate with each other. If a common interface is lacking, the developer is in charge of manually gluing the algorithms together, a tedious and error-prone operation. In addition, developers have to care about many other infrastructural tasks, like error checking, parallelization, and visualization.

A common approach in data processing is to define a framework that standardizes the interface of the algorithms with one another and offers some of the functionalities mentioned above. For the Python programming language alone several machine learning frameworks are available, including Orange [1], PyBrain [2], and the recent scikits.learn library.[2] One resulting problem is the overlap in the effort of providing a wide range of algorithms between alternative frameworks. Since different frameworks define different interfaces and features, common algorithm are often re-implemented, leading to an inefficient duplication.

In this paper, we discuss our efforts in the design of an open source framework for data processing algorithms, the Modular

* Corresponding author at: Institute for Theoretical Biology, Humboldt-Universität zu Berlin, Germany. Tel.: +49 089 37956958.
E-mail address: mail@nikowilbert.de (N. Wilbert).

[1] In general, data processing tasks range from low-level aspects of data analysis (e.g., the implementation of signal processing and machine learning algorithms) up to the high-level workflow management (possibly database integration and provenance). In this paper, we will use the term more restrictively to mean the former, i.e., data analysis tasks like filtering, transformation, and classification of data. Examples of more general workflow frameworks in Python are VisTrails (http://www.vistrails.org/), the Enthought Tool Suite (http://www.enthought.com/products/ets.php) (ETS) and the recent Sumatra (http://neuralensemble.org/trac/sumatra) toolkit. Beyond the Python Ecosystem there are also workflow tools like Kepler Project (https://kepler-project.org/) and Taverna (http://www.taverna.org.uk/).

[2] http://scikit-learn.sourceforge.net/.

1877-7503/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.jocs.2011.10.005

toolkit for Data Processing (MDP).[3] MDP offers a common interface to combine algorithms according to a pipeline analogy, making it possible to build complex data processing software in a modular fashion. In MDP we have tried to make the bookkeeping functionality, like automatic casting to numerical types and checks for the consistency in the dimensionality of the data, transparent for algorithm development. Developers wanting to add a new data-processing method in the MDP framework usually only need to implement the core algorithm, while the rest is transparently handled by MDP. As a result, MDP has been adopted by several other, more specialized libraries, for instance PyMVPA [4], PyMCA [5], OpenElectrophy [6], and the Oger toolbox.[4]

To maximize code reuse, our long-term philosophy is to automatically wrap algorithms defined in external libraries if they are available. For example, MDP provides wrappers for LIBSVM [7] and Shogun [8], and we are experimenting with a scikits.learn support. This should enable users to seamlessly combine the algorithms defined in MDP with the functionality in those libraries. To keep the library interface concise, the implementation of the core library remains relatively lightweight and new features like parallelization and caching are added using an extension mechanism inspired by aspect-oriented programming practices.

The paper proceeds as follows: in the next two sections we summarize the basic concepts and building blocks of the library, including an overview of a recent addition to the framework called BiMDP, which allows implementing general feedback processes like error back-propagation and gradient descent. Next, we discuss an MDP extension that provides functionality to define parallelized algorithms. This feature is based on a general extension mechanism that is analyzed in a dedicated section. We conclude with a real-life example of an application that performs handwritten digit recognition using MDP.

2. Basic elements of MDP: Nodes and Flows

The core of MDP consists of a collection of supervised and unsupervised learning algorithms and other data processing units that are encapsulated in *nodes* with a standardized interface. Multiple nodes can be combined into data processing sequences, which are called *flows*. The base of available algorithms in MDP is steadily increasing and includes signal processing methods (e.g., Principal Component Analysis (PCA), Independent Component Analysis (ICA) algorithms like FastICA [9], Slow Feature Analysis (SFA) [10]), manifold learning algorithms (e.g., Locally Linear Embedding (LLE) [11], Growing Neural Gas (GNG) [12]), various classification methods, and many others. A detailed description of these basic elements can be found in [13]. In the following we summarize the most salient features and describe the elements of the library that have been introduced only recently.

Most data processing algorithms can be divided in two distinct phases: in the first phase a set of training data is used to adjust a set of internal parameters (the *training phase*); in the second phase the algorithm makes use of the learned parameters to process test data (the *execution phase*). This basic structure is mirrored in MDP using an object oriented design, where algorithms are represented by classes derived from a `Node` base class. The input and output data for a node is encapsulated in a 2D NumPy array, representing on the first axis the different data points ("samples"), and on the second the (potentially multi-dimensional) data values. Each node object is characterized by the dimensionality of the input and output data, and the type of the data in the array (potentially arbitrary but usually a numerical type like single and double precision floating-point numbers). By default, these attributes are inherited from the input data. For algorithms that require supervised training it is also possible to provide additional data during training (e.g., the class labels in the example of Section 6).

Node objects are finite state machines; if the algorithm requires one or more training phases, the node stays in a training state until instructed by the user to proceed to the execution state. Nodes with no training phase at all are also possible (e.g., the polynomial expansion node used in Section 6). As an example, consider the `PCANode`, an implementation of the Principal Component Analysis (PCA) algorithm. PCA reduces the dimensionality of the input data by preserving only the directions with maximal variance[5] [14]:

```
import mdp
import numpy as np
# create dummy data: 50 data points,
10-dimensional
data = np.random.random((50,10))
# create PCANode instance; reduce data
dimensionality to 5
node = mdp.nodes.PCANode(output_dim=5)
# train algorithm
node.train(data)
node.stop_training()
# project data onto the five principal components
proj_data = node.execute(data)
```

The `Node` base class has been designed to allow arbitrarily large sets of data by accepting input data in multiple chunks. It is thus possible to perform computations with data that would not fit into memory, or to generate the training data on-the-fly as needed:

```
# iterate over training data chunks, which must be
arrays
for data_chunk in data_source:
  node.train(data_chunk)
node.stop_training()
```

This works for all data sources that are *iterables* (i.e., which can be iterated over, as for a list of data chunks).

To make writing new nodes as convenient as possible for algorithm implementors, the `Node` base class takes care of issues like error checking, automatic casting, and so on. Typically, developers only have to override the private methods `_train`, `_stop_training`, and `_execute` to implement their algorithm.[6] A public version of these methods (without the underscore) is provided by the `Node` base class. These template methods perform all of the tasks related to the framework, and then delegate to the private methods for the actual computations.

Nodes can be combined into data processing sequences, or *flows*. Given a set of input data, flows take care of successively training and executing all nodes in a sequence. This allows the user to construct complex algorithms as a series of simpler data processing steps. In

[3] MDP is available at http://mdp-toolkit.sourceforge.net/, and is licensed under the BSD open-source license. This publication discusses MDP version 3.2, which is compatible with Python ≥2.5 (including Python 3). MDP relies on the NumPy and SciPy libraries [3] for fast numerical matrix or array operations, with NumPy being the only mandatory dependency. A comprehensive tutorial and different usage examples are available on the MDP homepage.

[4] http://www.reservoir-computing.org/organic/engine.

[5] The explicit training in this example is not strictly required, because calling `execute` on an untrained node automatically triggers the training. It is thus possible to summarize the last four lines as `proj_data = mdp.nodes.PCANode(output_dim=5)(data)`. This user-friendly functionality is implemented in the node base class.

[6] In the Python convention, a leading underscore indicates that the method is private, only intended for internal use by the class.

the following example two nodes are created, trained, and executed in a flow[7]:

```
# create two nodes
pca_node = mdp.nodes.PCANode(output_dim=10)
sfa_node = mdp.nodes.SFANode()
# combine them in a flow
flow = mdp.Flow([pca_node, sfa_node])
# or alternatively: flow = pca_node +sfa_node
# train nodes with different data sources
flow.train([data_source1, data_source2])
# pipe some data through the fully trained flow
y = flow.execute(data_source3)
```

For each node there must be a data source providing the training data for that node (a data source can be any kind of iterable object). Note that the training data for a node is first processed by the previous, already trained nodes. For example the data from `data_source2` are executed by `pca_node`, and the resulting output is then used for the training of `sfa_node`. The flow is responsible for managing the training and execution of nodes, automating these standard tasks.

Using `Flow`s, users can create data processing applications as a linear sequence of algorithms. The sub-package `mdp.hinet` extends the capabilities of flows to arbitrarily complex, hierarchical, feed-forward architectures, providing for example nodes that define layers of algorithms. This has been used to create large multilayer networks for object recognition [15] and other tasks.

3. Bidirectional data flow: BiMDP

The MDP library as described so far executes algorithms in a strictly feed-forward manner. Data passes through the nodes in the order specified at the flow construction time. This has some limitations: for example, there is no built-in support for error back-propagation [16,17], which is needed for neural networks and other deep learning architectures [18]. Input and output data in MDP is generally also restricted to 2D arrays, which can be inconvenient for some applications. BiMDP is a recent, major addition to the MDP framework that addresses this variety of use cases.

The `bimdp` package expands the feed-forward flow processing of MDP with the ability to transfer diverse data and execute nodes in arbitrary order. This makes it possible to use the MDP framework for a larger class of algorithms, like gradient descent algorithms or Deep Belief Networks (DBN) [19]. The prefix "bi" stands for *bidirectional*, i.e., enabling data to travel both up and down a flow. Nevertheless, BiMDP stays as close as possible to standard MDP. It provides alternative, extended versions of the `Node` and `Flow` classes and of subpackages like `hinet`. The most important features in BiMDP are:

- Nodes can specify other nodes as *targets* where execution will continue. This is realized by the `bimdp.BiFlow` class, which replaces the standard `Flow` class in BiMDP, and extends it with additional functionality, e.g. the possiblity to define feed-back processes and loops. The complexities of arbitrary data flow patterns are evenly split between `BiNode` and `BiFlow`: nodes specify their data and target using a standardized interface, which are then interpreted by the flow in a way analogous to a primitive domain specific language with a "goto" statement.
- In addition to the standard array data, nodes can transport arbitrary data in a message dictionary. The new `BiNode` base class provides functionality to make this as convenient as possible, by automatically extracting the data for each node. For example, when training a classifier algorithm (as in the example of Section 6, Fig. 1), the label data can be stored in a message under the `"label"` key. The label data is then automatically extracted when the classifier node is reached. Additional functionality allows addressing specific nodes using special message keys.
- An interactive HTML-based inspection tool for flow training and execution is provided. This allows users to step through the flow node by node for graphical debugging or analysis purposes. Custom visualizations can be integrated as well (e.g., in the form of data plots for intermediate data).

[7] `SFANode` is an implementation of the Slow Feature Analysis algorithm, which extracts the most slowly-varying features from the input signals [10].

```
import mdp
import mnistdigits  # helper module for digit dataset

# Create the nodes and combine them in a flow
flow = mdp.parallel.ParallelFlow([
            mdp.nodes.PCANode(output_dim=40),
            mdp.nodes.PolynomialExpansionNode(degree=2),
            mdp.nodes.FDANode(output_dim=(mnistdigits.N_IDS-1)),
            mdp.nodes.GaussianClassifier(execute_method="label")
       ])

# Prepare training and test data.
train_data, train_ids = mnistdigits.get_data("train")
train_labeled_data = zip(train_data, train_ids)
train_iterables = [train_data, None,
                   train_labeled_data, train_labeled_data]
test_data, test_ids = mnistdigits.get_data("test")

# Parallel training and execution.
with mdp.parallel.ProcessScheduler() as scheduler:
    flow.train(train_iterables, scheduler=scheduler)
    result_labels = flow.execute(test_data, scheduler=scheduler)

# Analysis of the results.
n_samples = 0
n_hits = 0
for i, id_num in enumerate(test_ids):
    chunk_size = len(test_data[i])
    chunk_labels = result_labels[n_samples:(n_samples+chunk_size)]
    n_hits += (chunk_labels == id_num).sum()
    n_samples += chunk_size
print "performance: %.1f%%" % (100. * n_hits / n_samples)
```

Fig. 1. Example code for MNIST digit recognition. A detailed description is given in Section 6. This code achieves about 96.9% correct classification on the test set.

The approach taken by BiMDP is very general and allows the execution of algorithms in an arbitrary sequence, making it possible to use the established features of MDP in situations that were previously not supported.

4. Parallelization

Some of the algorithms implemented in MDP could in principle be parallelized in a straightforward way. For example, the PCA algorithm is based on computing the covariance matrix of the data. It is possible to split the task of computing the covariance matrix for a large data set in independent blocks by dividing the data into several subsets and recording the second moment matrix, the mean, and the number of samples individually for each subset. From this statistics, the covariance matrix of the whole data set can easily be reconstructed. The calculations for each block are independent from each other and can therefore be performed in parallel. Problems like this are called *embarrassingly parallel*: a negligible effort is needed to segment the problem into tasks that can be executed independently, with a speedup proportional to the number of threads [20].

Parallelization is an increasingly important topic, since progress in the speed of a single CPU core has slowed down in recent years. Instead, Moore's law is now perpetuated by providing an increasing number of CPU cores [21], which require parallelization to be used

effectively. The `parallel` package in MDP adds functionality to parallelize the training and execution of MDP flows, thus providing a convenient speedup that scales with the number of CPU cores.

In the last few years, there has also been a huge trend to use Graphics Processing Units (GPU) for general numerical applications, due to their highly data-parallel number crunching abilities. GPU support in Python is for example provided by PyCUDA [22] and Theano [23]. The parallelization approach that we discuss here is complementary to this and could in principle also be used to distribute work across multiple GPUs (in a single machine or cluster).

4.1. The `parallel` package

The `parallel` package in MDP is divided into two weakly coupled parts:

- The first part consists of *schedulers*. A scheduler takes arbitrary "tasks" and processes them in parallel (e.g., in multiple Python processes). A scheduler deals with the more technical aspects of parallelization but does not need to know about nodes and flows, as it uses a different level of abstraction. Schedulers in MDP are derived from the `Scheduler` base class, which defines a unified interface for all the different scheduler types, enabling MDP to work with any scheduler implementation. This approach also makes it easy to write MDP adapters for the numerous scheduler solutions that are available. The standard scheduler in MDP is the `ProcessScheduler`, which distributes the incoming tasks over multiple Python processes. The use of processes instead of threads is sometimes necessary due to a current limitation of the standard Python Virtual Machine (VM), which is not thread-safe. Therefore, a global mechanism is used to prevent threads from running concurrently (the *global interpreter lock*, or GIL). However, libraries like NumPy release the GIL while running compiled code outside the Python VM. Therefore, it is often possible to get a speedup even when using Python threads, and the parallelization overhead is generally lower with this approach (largely thanks to shared memory, avoiding data transfer between processes). MDP provides a `ThreadScheduler` class to take advantage of this. Depending on the situation it can be much faster than the `ProcessScheduler`. In addition, MDP has experimental support for the Parallel Python library,[8] with which it is possible to distribute tasks across multiple machines.
- The second part consists of parallel versions of the regular MDP nodes and flows. All MDP nodes naturally support parallel execution, since copies of a node can be made and executed in parallel. Parallelization of the training phase, on the other hand, depends on the specific algorithm. If a node class supports parallel training (e.g., `PCANode`), it provides a `fork` and a `join` method. The `fork` method is used to create multiple instances ("copies") of the node for parallel training. When the training of these instances is done they are combined using the `join` method (e.g., for the `PCANode` this combines the per-chunk data to build the overall covariance matrix). The extension mechanism of MDP is used to provide the `fork` and `join` implementations, as explained in Section 5. MDP provides parallel implementations of a subset of commonly used nodes.

For the convenient parallel execution or training of a flow, the `ParallelFlow` class is used instead of the normal `Flow`. This class takes care of managing the nodes, providing tasks to a scheduler, and combining the results. In the following example, the training of a simple flow (the example of Section 2) is parallelized using multiple CPU cores:

```
node1 = mdp.nodes.PCANode(output_dim=10)
node2 = mdp.nodes.SFANode()
pflow = mdp.parallel.ParallelFlow([node1, node2])
with mdp.parallel.ProcessScheduler() as scheduler:
  pflow.train(data_sources, scheduler=scheduler)
```

Only two small changes were needed to parallelize the training of the flow: `ParallelFlow` is used instead of the normal `Flow` class, and a scheduler was passed to the `train` method. The `ProcessScheduler` automatically creates as many Python processes as there are CPU cores. Then the parallel flow creates a training task for each data chunk returned by the `data_sources` and hands it over to the scheduler, which distributes them across the available worker processes. The scheduler context manager (which is invoked by the Python `with` statement) ensures that the scheduler is correctly shut down after training, even in the case of an exception.

4.2. Performance

The real-world performance gain for this parallelization approach depends on the specific application. Originally, the parallel package was created for large, hierarchical models for the processing of sequences of images [15,24]. The training of a network of these dimensions can take several hours and is ideally suited for parallelization, as the overhead for transporting data between processes or machines is small compared to the computational cost of each task. Therefore, the process-based scheduler scales nicely with the number of CPU cores. In simulations, we measured a speed-up factor of 4.2 on an Intel Core i7 920 processor with 4 physical/8 logical cores, using 8 processes. For other models, like the example presented later in Section 6, the speedup is typically lower but still worthwhile, since the cost of parallelizing an application in this way is very low. This approach slightly contradicts the (generally rightful) view of parallelization as a last resort in optimization. However, by making parallelization easy we hope to give the scientific developer this option, as a complement to other optimization efforts (like, for example, using Cython [25] to write faster code).

5. Node extensions

The node extension mechanism addresses a problem that is quite common in software design and is well illustrated by the situation in the `parallel` package: in order to add the parallel training feature, the new `fork` and `join` methods must be made available. The straightforward solution would be to add these methods in the original class definitions, since they are class-dependent and require access to internal data (e.g., in the case of PCA the covariance matrix). However, parallelization is not the only node feature that has been added over time. Other examples in MDP are the HTML-based node visualization [13] and the new caching mechanism. Whenever a new feature is added, the original code would need to be modified across many different modules. As more features are added, the classes would grow larger, and the dependencies would pile up. This solution does not scale well, and it violates standard software design principles.

An alternative approach would be to add the new features through inheritance, deriving new node classes that implement the feature for the parent node class. This requires that the right class is picked when the node is instantiated, depending on the required feature. For example, to use parallelization with the `PCANode`, instances of the `ParallelPCANode` class would have to be used. This leads to problems when the use of multiple

[8] http://www.parallelpython.com.

new features is required. Classes would have to be created for every combination of features that might be used (e.g., a `Parallel_HTML_PCANode` class). Again, this approach scales poorly with the number of different features. The solution to these issues was finally found in the node extension mechanism.

5.1. Node extension mechanism and aspect-oriented programming

The node extension mechanism adds methods and class attributes to node classes during runtime, thereby dynamically adding new features. In this way, extensions can be activated exactly when they are needed, reducing the risk of interference with other extensions. It is possible to use multiple extensions at the same time as long as there are no name collisions. Users can easily implement custom extensions, thus adding new functionalities to MDP nodes without modifying the code of the library.

The node extension mechanism enables a mild form of aspect-oriented programming (AOP) [26] in order to deal with the so-called *cross-cutting concerns*. In the AOP terminology the new features are *aspects*. The problem is that even though these aspects are typically very focused they do affect many node classes spread all over the MDP code base. The term "cross-cutting" means that they alter a large amount of otherwise separated code (orthogonal to the organization of that code). Following the AOP terminology, the methods that are added contain *advice*. These ideas are not new in the Python world, and solutions that are similar to the MDP extension mechanism have been implemented before.[9]

5.2. Using extensions

Extensions in MDP are activated via the `activate_extension` function. For example activating the parallel extension is as simple as:

```
node = mdp.nodes.PCANode()
hasattr(node, 'fork') # returns False
mdp.activate_extension('parallel')
# now the added attributes / methods are available
hasattr(node, 'fork') # returns True
mdp.deactivate_extension('parallel')
# the additional attributes are no longer
available
hasattr(node, 'fork') # False again
```

Activating an extension adds the available extension attributes to the supported nodes.[10] MDP also provides an extension context manager for the `with` statement:

```
with mdp.extension('parallel'):
  hasattr(node, 'fork') # returns True
```

The `with` statement ensures that the activated extension is deactivated after the code block, even if there is an error. Finally, one can also use a function decorator to activate an extension for the time that the function is executed:

```
@mdp.with_extension('parallel')
def f(node):
  return hasattr(node, 'fork') # returns True
```

Both the context manager and function decorator only deactivate those extensions that were not already active when the context was entered, to avoid unintended side effects.

In addition to the previously mentioned `parallel` and `HTML` extensions there are a couple of other available extensions. The most recent additions in MDP are the `gradient` extension, which allows calculating the overall gradient for a sequence of nodes, and the `cache_execute` extension, used to cache node results for greater efficiency and based on the `joblib` library.[11]

5.3. Writing extensions

Adding extension methods to a node class can be done in two ways: via multiple inheritance or via a function decorator. For the first approach, we derive from both the extension base class and from the node class. For example, the parallel extension of the PCA node looks like this:

```
class ParallelPCANode(ParallelExtensionNode,
                mdp.nodes.PCANode):
  def _fork(self):
    # implement the forking for PCANode
    pass
  def _join(self):
    # implement the joining for PCANode
    pass
```

`ParallelExtensionNode` is the base class of the extension. The required methods or class attributes are defined just like in a normal class. In principle, one could use the `ParallelPCANode` class like a normal class, but with the extension mechanism the `ParallelPCANode` class never has to be instantiated. Instead, the methods defined here are automatically registered with the `PCANode` and the parallel extension.

Alternatively, one can use the `extension_method` function decorator to create extension methods. The extension method can be defined like a normal function, with the function decorator added on top. To create the `_fork` method for the `PCANode` one could write:

```
@mdp.extension_method('parallel', mdp.nodes.PCANode)
  def _fork(self):
    # implement the forking for PCANode
    pass
```

The first decorator argument is the name of the extension, the second is the class to which it belongs. Using the decorator instead of the class-based approach is convenient when only a single extension method is added per node class (but technically the two approaches are equivalent).

5.4. Creating extensions

To create a new node extension one simply has to define a new extension base class. For the parallel extension this starts like:

```
class ParallelExtensionNode(mdp.ExtensionNode,
mdp.Node):
    extension_name = 'parallel'
    def _fork(self):
        raise NotImplementedError()
```

When defining a new extension one always has to define the `extension_name` attribute. This name can then be used to activate or deactivate the extension. When inheriting from `ExtensionNode`, a special meta-class is used for the class creation, which stores all the defined class attributes in a centralized extension dictionary, indexed by the extension name and the node class name. The `mdp.extension_method` decorator works in a similar way, storing a defined function in the same central extension dictionary. Whenever an extension is activated, the extension dictionary is accessed to add the stored extension attributes to the respective node classes at runtime.

[9] See for example http://www.cs.tut.fi/ ask/aspects.

[10] In this case, the explicit activation of the parallel extension is typically not needed, since this is handled by the `ParallelFlow` class.

[11] http://packages.python.org/joblib/.

6. Example application: handwritten digit recognition

In this section, we present a complete application that makes use of several features of MDP (Fig. 1). Nonlinear Fisher Discriminant Analysis (FDA) [27] is used to classify handwritten digits from the well known MNIST data set, following the method presented in [28]. This is a classification problem with $C = 10$ different classes (one for each digit), and there are about 6000 training and 1000 test samples per class. The example code is also available on the MDP homepage [12] together with some other variants (e.g., a version using BiMDP).

The first step in this program is the creation of the required node instances, which are directly combined in a flow (lines 5–10). The first node performs PCA to reduce the dimensionality of the data from 784 (each digit image has $28 \times 28 = 784$ black or white pixels) to 40. Next, we perform a quadratic expansion (i.e., expand the input data with elements that are quadratic monomials of the original components, like $x_1 x_2$ or ${x_1}^2$), which increases the dimensionality to 860. Then, Fisher Discriminant Analysis (FDA) is performed to reduce the dimensionality again to $C - 1$ components. FDA maximizes the inter-class variance vs. the intra-class variance, ideally forming a separate cluster for each class. The last node is a Gaussian classifier, which has the task of identifying the class clusters. The `execute_method` argument instructs the classifier node to return the most likely classification label when it is executed (as opposed to, for example, the probability of each label).

In the second part of the code, we load the MNIST digits data via a simple helper module (called `mnistdigits` and included in the MDP examples online; line 13). The module returns two lists called `train_data` and `train_ids`. `train_data` consists of 10 elements, each being an array containing all the training images for that digit class. These arrays are two-dimensional: the first index is the sample number, the second one indexes the pixels. The `train_ids` list contains a list of class labels (0–9) for the training data. Since `Flow` requires that the training data for each node is provided by a single iterable, we use the built-in `zip` function to combine the two lists into a single list consisting of tuples with two elements: the data array and the corresponding class label (line 14). At lines 15–16, we define a list of data sources, one for each node, to be used for training. The entries in this list vary, as for example the PCA node does not require class labels, and the quadratic expansion node requires no training at all.

Once the flow is defined, we proceed with training the nodes and analyzing the output of the application on test data. In this example, the process-based scheduler is used to speed up the training. Using Python's `with` statement we create a scheduler instance (line 20) using as many processes as there are CPU cores. Calling the `train` method is then enough to automatically perform the parallel training. The test data is processed in parallel as well. We can use this example as a benchmark for the parallelization speedup. For such a simple example the overhead of the parallelization is still significant, so one cannot expect the performance to scale linearly with the number of cores. The benefits of parallel execution are also reduced by load-balancing issues due to the low number of data chunks (10 uneven chunks distributed across 4 processes). On a four core machine we measured a parallel speedup factor of about 2.1 (29 s vs. 62 s running time for training and testing). The thread-based scheduler is slightly faster and achieves a factor of 2.5 with 4 threads.

[12] http://mdp-toolkit.sourceforge.net/.

7. Conclusion

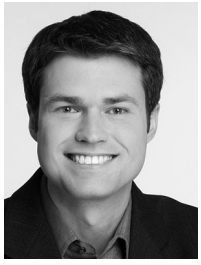
Across many fields in scientific research there has been over the past few years an increasing readiness to share source code. We believe that this is a very positive trend and that it will lead to greater transparency and faster development cycles in the future. An equally important step toward simplifying the daily routine of researcher, though, is to provide them with software frameworks that free them from repetitive programming patterns.

In this paper we described the tools that we have developed in the MDP framework to automatize common tasks in data processing applications, for example to manage complex sequences of algorithms and execute them in parallel. We also discussed the design choices we have made in MDP in order to avoid some of the most common problems in this kind of framework, in particular the MDP extension mechanism, with which the algorithms can be extended dynamically with new features, avoiding an explosion of the framework's interface. Hopefully, our experiences will be useful in other contexts and, together with other current attempts at finding design patterns in scientific programming, will lead to cleaner and more efficient scientific code.

References

[1] J. Demšar, B. Zupan, G. Leban, T. Curk, Orange: from experimental machine learning to interactive data mining, Knowledge Discovery in Databases: PKDD 2004 (2004) 537–539.
[2] T. Schaul, J. Bayer, D. Wierstra, Y. Sun, M. Felder, F. Sehnke, T. Rückstieß, J. Schmidhuber, Journal of Machine Learning Research 11 (2010) 743–746.
[3] E. Jones, T. Oliphant, P. Peterson, et al., SciPy: Open Source Scientific Tools for Python, 2001–present, http://www.scipy.org/.
[4] M. Hanke, Y. Halchenko, P. Sederberg, E. Olivetti, I. Fründ, J. Rieger, C. Herrmann, J. Haxby, S. Hanson, S. Pollmann, PyMVPA: a unifying approach to the analysis of neuroscientific data, Frontiers in Neuroinformatics 3 (0), doi:10.3389/neuro.11.003.2009.
[5] V. Solé, E. Papillon, M. Cotte, P. Walter, J. Susini, A multiplatform code for the analysis of energy-dispersive X-ray fluorescence spectra, Spectrochimica Acta Part B: Atomic Spectroscopy 62 (1) (2007) 63–68.
[6] S. Garcia, N. Fourcaud-Trocmé, OpenElectrophy: an electrophysiological data-and analysis-sharing framework, Frontiers in Neuroinformatics 3, doi:10.3389/neuro.11.014.2009.
[7] C.-C. Chang, C.-J. Lin, LIBSVM: a library for support vector machines, ACM Transactions on Intelligent Systems and Technology 2 (2011), 27:1–27:27.
[8] S. Sonnenburg, G. Ratsch, F. De Bona, The SHOGUN machine learning toolbox, Journal of Machine Learning Research 11 (2010) 1799–1802.
[9] A. Hyvärinen, Fast and robust fixed-point algorithms for independent component analysis, IEEE Transactions on Neural Networks 10 (1999) 626–634.
[10] L. Wiskott, T. Sejnowski, Slow feature analysis: unsupervised learning of invariances, Neural Computation 14 (4) (2002) 715–770.
[11] S. Roweis, L. Saul, Nonlinear dimensionality reduction by locally linear embedding, Science 290 (5500) (2000) 2323.
[12] B. Fritzke, A growing neural gas network learns topologies, Advances in Neural Information Processing Systems (1995) 625–632.
[13] T. Zito, N. Wilbert, L. Wiskott, P. Berkes, Modular toolkit for Data Processing (MDP): a Python data processing framework, Frontiers in Neuroinformatics 2, doi:10.3389/neuro.11.008.2008.
[14] I. Jolliffe, Principal Component Analysis, Springer-Verlag, 1986.
[15] M. Franzius, N. Wilbert, L. Wiskott, Invariant object recognition and pose estimation with slow feature analysis, Neural Computation 23 (9) (2011) 2289–2323.
[16] A.E. Bryson, Y.-C. Ho, Applied Optimal Control: Optimization, Estimation, and Control, Blaisdell Publishing Company, 1969.
[17] D. Rumelhart, G. Hinton, R. Williams, Learning representations by back-propagating errors, Nature 323 (1986) 533–536.
[18] Y. Bengio, Learning Deep Architectures for AI, Now Publishers Inc., 2009.
[19] G. Hinton, S. Osindero, Y. Teh, A fast learning algorithm for deep belief nets, Neural Computation 18 (7) (2006) 1527–1554.
[20] I. Foster, Designing and Building Parallel Programs, Addison-Wesley, 1995.
[21] H. Sutter, The free lunch is over: a fundamental turn toward concurrency in software, Dr. Dobb's Journal 30 (3) (2005) 16–20.
[22] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, A. Fasih, Pycuda GPU runtime code generation for high-performance computing, CoRR abs/0911.3456.
[23] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, Y. Bengio, Theano: a CPU and GPU math expression compiler, in: Proceedings of the Python for Scientific Computing Conference (SciPy), 2010, oral, http://www.iro.umontreal.ca/lisa/pointeurs/theano_scipy2010.pdf.

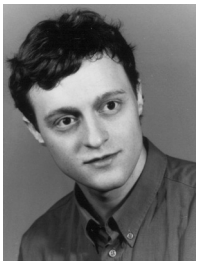
[24] M. Franzius, H. Sprekeler, L. Wiskott, Slowness and sparseness lead to place, head-direction, and spatial-view cells, PLoS Computational Biology 3 (8) (2007) e166.
[25] D. Seljebotn, Fast numerical computations with Cython, in: Proceedings of the 8th Python in Science Conference, 2009.
[26] G. Kickzales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, J. Irwin, Aspect-oriented programming, in: Proceedings of ECOOP, IEEE, Finland, 1997, pp. 220–242.
[27] R. Fisher, et al., The use of multiple measurements in taxonomic problems, Annals of Eugenics 7 (2) (1936) 179–188.
[28] P. Berkes, Handwritten digit recognition with nonlinear Fisher discriminant analysis, in: Proceedings of ICANN 2005 2 (LNCS 3696), 2005, pp. 285–287.



Niko Wilbert is currently finishing his PhD in the computational neuroscience research group of Laurenz Wiskott in Berlin (Germany). His research interests include models of the mammalian visual system and machine learning. He has a diploma degree in physics from the RWTH Aachen (Germany) and a MSc from the Imperial College London (UK).

Tiziano Zito studied theoretical physics at the University of Bologna (Italy) and wrote his master thesis about the charge distribution of polymer chains. He pursued his PhD in computational neuroscience at the Institute for Theoretical Biology of The Humboldt University in Berlin (Germany) with a thesis about the use of the slowness principle in models of the auditory system. He currently works as a scientific computing consultant in the Modelling of Cognitive Processes group at the Technical University in Berlin.

Rike-Benjamin Schuppner was born in 1981 in Marburg, Germany and is now living in Berlin. He studied physics in Marburg and Berlin where he received his diploma in 2009 working on numerical simulations with neuronal models. After that, he started working as a programmer. Since 2009 he is also contributing to the MDP core.



Zbigniew Jędrzejewski-Szmek received a MSc in molecular spectroscopy at the University of Warsaw. Currently he is writing a PhD dissertation which includes the creation of Python software for measurement analysis and modelling of molecules. He is a developer in two open-source neuroscience projects.

Laurenz Wiskott studied physics in Göttingen and Osnabrück, Germany, and received his PhD in 1995 at the Ruhr-Univerity Bochum. The stages of his career include three-years at The Salk Institute in San Diego, one year at the Institute for Advanced Studies in Berlin, and nine years at the Institute for Theoretical Biology, Humboldt-University Berlin, where he was heading a junior research group and became professor in 2006. Since 2008 he is at the Institute for Neural Computation, Ruhr-Univerity Bochum. He has been working in the fields of Computer Vision, Neural Networks, Machine Learning and Computational Neuroscience.

Pietro Berkes is a postdoctoral research fellow at the visual information processing and learning laboratory at Brandeis University, MA, USA. He graduated PhD in biophysics from Humboldt University, Berlin, Germany, in 2005, and continues his training in computational neuroscience and machine learning at the Gatsby Computational Neuroscience Unit, London, UK, 2005–2008. His research interests concern the neural computations underlying sensory processing and learning.