Modular Toolkit for Data Processing (MDP)

Niko Wilbert[1,2], Tiziano Zito[2,3], Rike-Benjamin Schuppner[2], Zbigniew Jędrzejewski-Szmek[4], Laurenz Wiskott[1,2,5], Pietro Berkes[6]

1 Institute for Theoretical Biology, Humboldt-Universität zu Berlin, Germany
2 Bernstein Center for Computational Neuroscience, Berlin, Germany
3 Modelling of Cognitive Processes, Berlin Institute of Technology, Germany
4 Institute of Experimental Physics, Warsaw University, Poland
5 Institut für Neuroinformatik, Ruhr-Universität Bochum, Germany
6 National Volen Center for Complex Systems, Brandeis University, USA

Abstract

Data processing is an ubiquitous task in scientific research, and much energy is spent in the development of appropriate algorithms. As a consequence, it is relatively easy to find an implementation of the most common methods. However, any typical data processing application requires the combination of several such algorithms, or the application of a single one repeatedly for different subsets of the data (and often both). Users are thus often forced to manually patch different function interfaces together, and to manually manage the required data transformations. 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. The modularity of MDP also has enabled the addition of new capabilities, like parallelism, an extension mechanism, and support for algorithms that require bi-directional data flow. In this paper, we discuss the design of the library and how it permits developers to write efficient, re-usable code.

1 Introduction

A typical data processing application requires the application of several algorithms to achieve the desired result. For example, a face recognition application might require to filter the incoming images, reduce the dimensionality of the data, and classify the resulting high-dimensional vectors as one of the persons in the database. Although each one of the processing steps might be a well-known method, and as such likely to be available in some form online, for the complete application these methods also need to be able to communicate with each other. If a common interface is lacking, the developer will be in charge of manually gluing the algorithms together, a tedious and error-prone operation.

The Modular toolkit for Data Processing (MDP)[1] is an open source Python library of widely used data processing algorithms, and offers a framework to combine them according to a pipeline analogy. This makes it possible to build complex data processing software in a modular fashion. MDP

[1] `http://mdp-toolkit.sourceforge.net/`

alleviates the problems mentioned above, in that developer can concentrate on implementing the learning and execution steps of a new algorithms, while the framework defines a common interface, performs consistency checks, and offers additional services like parallelization. As a consequence, the resulting code is robust and re-usable. MDP is easily embedded or used by other libraries (current examples are PyMVPA[2] and PyMCA[3]).

MDP shares common goals with other Python machine learning libraries like Orange[4] and PyBrain[5], even though these differ in their design approach and focus on different sets of algorithms. Also, there exist other collections of algorithms, like for example the recent scikits.learn[6] package. To maximize code reuse, our long-term philosophy is to automatically wrap the algorithm defined in these libraries if they are available. For example, MDP provides wrappers for LIBSVM [1] and Shogun [2]. Users can thus seamlessly combine the algorithms defined in MDP with the functionality defined in those libraries.

In the next section, we will discuss the basic design of the library. The modularity of MDP also has enabled the addition of new capabilities, like parallelism and an extension mechanism. The most recent upgrade allows MDP to implement general feed-back processes like error backpropagation and gradient decent. These new features are introduced in Section 3.

2 Nodes and Flows

The core of MDP is a collection of supervised and unsupervised learning algorithms and other data processing units that are encapsulated in *nodes* with a standardized interface. The `Node` class was designed to be applied to arbitrarily long sets of data: if the underlying algorithms supports it, the internal structures can be updated incrementally by sending multiple batches of data. It is thus possible to perform computations on amounts of data that would not fit into memory or to generate data on-the-fly. The general form of the training phase thus is (in this example we use a node implementing the Principal Component Analysis algorithm):

```
node = mdp.nodes.PCANode()
for data_batch in data_source:
   node.train(data_batch)
node.stop_training()
```

After the training is finished the trained algorithm can be applied to data using the `node.execute` method. The base of available algorithms in MDP

[2] `http://www.pymvpa.org/`
[3] `http://pymca.sourceforge.net/index.html`
[4] `http://www.ailab.si/orange/`
[5] `http://pybrain.org`
[6] `http://scikit-learn.sourceforge.net/`

is steadily increasing and includes signal processing methods (e.g., Principal Component Analysis [3], Independent Component Analysis algorithms like FastICA [4], Slow Feature Analysis [5]); manifold learning algorithms (e.g., Locally Linear Embedding [6], Growing Neural Gas [7]); various classification methods; and many others.

The nodes can be combined into data processing sequences (*flows*). Given a set of input data, MDP takes care of successively training or executing all nodes in the flow. This allows the user to construct complex algorithms as a series of simpler data processing steps. In the following example two nodes are created, trained and executed in a flow:

```
pca_node = mdp.nodes.PCANode(output_dim=10)
sfa_node = mdp.nodes.SFANode()
flow = mdp.Flow([pca_node, sfa_node])
flow.train([data_source1, data_source2])
y = flow.execute(x)
```

3 Additional MDP Components

Over time MDP has gained several components that extend the basic framework. Following the principle of modularity these components can be mixed as needed (e.g. enabling the parallelization of large hierarchical networks as used in [8]).

The first addition was the `hinet` subpackage, which adds components for the construction of hierarchical networks. This has been used, for instance, to create large multi-layer networks for object recognition [9]. Due to its modular design, one can implement general feed-forward graphs with this package. For the visualization of complex networks MDP offers the automatic generation of HTML based representations (to be viewed in a browser or embedded into a custom GUI).

3.1 Parallelization

The `parallel` sub-package offers a parallel implementation of basic nodes and flows, enabling convenient parallelization for embarrassingly parallel problems. The `parallel` package in MDP is divided into two weakly coupled parts:

- The first part consists of the schedulers. A scheduler takes "tasks" and processes them in a more or less parallel way (e.g. in multiple Python processes). The scheduler classes in MDP are derived from the `Scheduler` base class, which provides a simple unified interface for all kinds of different schedulers. Schedulers do not depend on nodes and flows and it is easy to write an adapter for external schedulers.

For example MDP supports the Parallel Python[7] library to parallelize across multiple machines.

- The second part consists of parallel versions of the familiar MDP nodes and flows. In principle all MDP nodes support parallel execution, since copies of a node can be made and used in parallel. Parallelization of the training on the other hand depends on the specific algorithm. If a node class supports parallel training (like the `PCANode`) it is supposed to provide a `fork` and a `join` method. The `fork` method is used to create multiple instances ("copies") of the node for parallel training. After the training of these instances has been done they are combined using the `join` method (e.g., for the `PCANode` this means that the covariance matrices are combined).

  For convenient parallel execution or training of a flow, MDP provides a `Flow` subclass, `ParallelFlow`, which takes complete care of managing the nodes, providing tasks to the scheduler, and combining the results.

In the following example the training of a simple flow consisting of PCA and SFA is parallelized, so that multiple CPU cores are used:

```
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 are needed to parallelize the training of the flow: one has to use a `ParallelFlow` instead of the normal `Flow` and a scheduler must be provided. The scheduler context manager ensures that the scheduler is correctly shut down after training. The `ProcessScheduler` will automatically create as many Python processes as there are CPU cores. The parallel flow hands a training task for each data chunk over to the scheduler, which distributes them across the available worker processes. The results are returned to the flow and combined.

3.2 Node Extension Mechanism

Adding new capabilities to nodes (e.g., the `fork` and `join` methods for parallelization) generally requires that code is somehow injected into the existing classes. Modifying the original code for that would violate established design principle. On the other hand the use of specialized derived classes would create scaling issues when the number of features and their combinations increases. Therefore, MDP introduced a *node extension mechanism*, so that new capabilities can be implemented separately and are activated as needed.

[7] `http://www.parallelpython.com`

This mechanism is based on ideas from aspect oriented programming and has been used to implement a diverse set of features, ranging from memoization to creating the HTML representations of nodes. It also allows users to add custom extensions as needed.

3.3 BiMDP

The MDP library as described so far executes flows in a strictly feed-forward manner. There is no built-in support for error backpropagation [10][11] or similar techniques (which are used in neural networks and other deep learning architectures). The most recent and largest addition to MDP is the `bimdp` package, which extends the purely feed-forward flow processing of MDP with the ability of bidirectional data transfer. This makes it possible to use the MDP framework for a much larger class of algorithms, like gradient decent or Deep Belief Networks [12] (example implementations are available). The `hinet` and `parallel` sub-packages are also supported in BiMDP. Here is a brief summary of the most important features in BiMDP:

- Nodes can specify other nodes as jump targets, where the execution will be continued. This is enabled by the new `BiFlow` class and makes it possible to implement loops or backpropagation, in contrast to the strictly linear execution of a normal MDP flow. The new `BiNode` base class adds a `node_id` attribute, which can be used as a label to target a node. The complexities of arbitrary data flow patterns are evenly split up between `BiNode` and `BiFlow`: Nodes specify their data and target (using a standardized interface), which are then interpreted by the flow (similar to a primitive domain specific language).

- 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.

- 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).

The approach taken by BiMDP is very general and allows to execute and train algorithms in an arbitrary sequence. It will allow us to include in MDP algorithms that do not fit a simple feed-forward description.

4 Conclusion

Since its last article publication in 2008 [13] MDP has gained significant new capabilities like the built-in parallelization, node extensions and BiMDP. These additions have widened the scope of MDP, offering a unique combination of well integrated features. At the same time users continue to benefit from the mature code base and the comprehensive documentation. Since its first release in 2004 MDP has become a popular Python scientific software and has been used in various research projects, resulting in more than a dozen citations from various groups.

References

[1] Chih-Chung Chang and Chih-Jen Lin. *LIBSVM: a library for support vector machines*, 2001. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[2] S. Sonnenburg, G. Ratsch, and F. De Bona. The SHOGUN Machine Learning Toolbox. *Journal of Machine Learning Research*, 11:1799–1802, 2010.

[3] I.T. Jolliffe. *Principal component analysis*. Springer-Verlag, 1986.

[4] A. Hyvärinen. Fast and robust fixed-point algorithms for independent component analysis. *IEEE Transactions on Neural Networks*, 10:626–634, 1999.

[5] L. Wiskott and T. Sejnowski. Slow feature analysis: unsupervised learning of invariances. *Neural Computation*, 14(4):715–770, 2002.

[6] S.T. Roweis and L.K. Saul. Nonlinear dimensionality reduction by locally linear embedding. *Science*, 290(5500):2323, 2000.

[7] B. Fritzke. A growing neural gas network learns topologies. *Advances in neural information processing systems*, pages 625–632, 1995.

[8] R. Legenstein, N. Wilbert, and L. Wiskott. Reinforcement Learning on Slow Features of High-Dimensional Input Streams. *PLoS Comput Biol*, 6(8):e1000894, 2010.

[9] M. Franzius, N. Wilbert, and L. Wiskott. Invariant object recognition with slow feature analysis. In Vera Kurková, Roman Neruda, and Jan Koutník, editors, *Proc. 18th Intl. Conf. on Artificial Neural Networks, ICANN'08, Prague*, volume 5163 of *Lecture Notes in Computer Science*, pages 961–970. Springer, September 2008.

[10] A. E. Bryson and Yu-Chi Ho. *Applied optimal control: optimization, estimation, and control*. Blaisdell Publishing Company, 1969.

[11] D.E. Rumelhart, G.E. Hinton, and R.J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[12] G.E. Hinton, S. Osindero, and Y.W. Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

[13] T. Zito, N. Wilbert, L. Wiskott, and P. Berkes. Modular toolkit for Data Processing (MDP): a Python data processing framework. *Frontiers in Neuroinformatics*, 2, 2008.