

Oger: Modular Learning Architectures For Large-Scale Sequential Processing

David Verstraeten
Benjamin Schrauwen
Sander Dieleman
Philemon Brakel
Pieter Buteneers

*Department of Electronics and Information Systems
Ghent University
Ghent, Belgium*

DAVID.VERSTRAETEN@UGENT.BE
BENJAMIN.SCHRAUWEN@UGENT.BE
SANDER.DIELEMAN@UGENT.BE
PHILEMON.BRAKEL@UGENT.BE
PIETER.BUTENEERS@UGENT.BE

Dejan Pecevski

*Institute for Theoretical Computer Science
Graz University of Technology
Graz, Austria*

DEJAN@IGI.TUGRAZ.AT

Editor: TBD

Abstract

Oger (OrGanic Environment for Reservoir computing) is a Python toolbox for building, training and evaluating modular learning architectures on large datasets. It builds on MDP for its modularity, and adds processing of sequential datasets, gradient descent training, several cross-validation schemes and parallel parameter optimization methods. Additionally, several learning algorithms are implemented, such as different reservoir implementations (both sigmoid and spiking), ridge regression, Conditional Restricted Boltzmann Machine (CRBM) and others, including GPU accelerated versions. Oger is released under the GNU LGPL, and is available from <http://organic.elis.ugent.be/oger>.

Keywords: Python, modular architectures, sequential processing

1. Introduction

The rapid growth of Machine Learning as a research area has generated an almost proportionate amount of toolboxes and libraries, each with its own specific focus or design. The Oger toolbox originated from the need to rapidly implement, investigate and compare complex architectures built from state-of-the-art sequential processing algorithms, and to apply these architectures to large real-world tasks. Rather than contribute yet another toolbox which reimplements many standard algorithms, one of our design choices for Oger was to incorporate existing packages where possible.

Because modularity was one of the key requirements for Oger, it has been based on the well known and widely used Modular Data Processing toolkit (MDP) (Zito et al., 2008), which provides this modularity in addition to a wide variety of machine learning algorithms. Oger uses a Node as its basic building block: a (optionally trainable) data processing algorithm. These nodes can then be combined into an arbitrary feedforward graph structure called a Flow. Much of the error- and typechecking is abstracted away

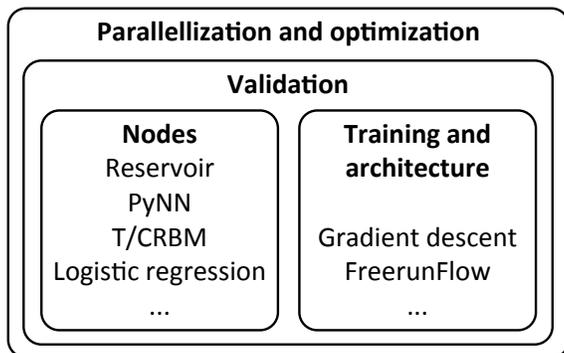


Figure 1: A schematic overview of the structure of Oger. The basic building blocks used for constructing learning architectures are used in a validation and optimization framework.

through the object-oriented interface, such that the developer can focus on implementing the actual algorithm.

Python was chosen as the development language because of several reasons. On the one hand it is a high-level, cross-platform and open-source interpreted language offering flexibility and rapid development, on the other hand interfaces to optimized numerical linear algebra packages such as BLAS and LAPACK are provided through the NumPy package so that the speed sacrifice remains limited. Mature and feature-complete packages for plotting (matplotlib) and general scientific computing (scipy) that in many respects come close to commercial alternatives are available, along with a plethora of smaller libraries providing specific functions. These reasons are likely also responsible for the growing popularity of Python within the scientific community in general.

2. Features

2.1 Nodes and algorithms

MDP implements several standard supervised and unsupervised learning methods for on stationary inputs, such as Principal Component Analysis, Independent Component Analysis, Slow Feature Analysis,¹ ... Oger adds several new nodes to this set:

- Several reservoir nodes (Verstraeten et al., 2007): a base node with customizable nonlinear function and weight topologies, a leaky integrator reservoir node, and a GPU-optimized reservoir using CUDA.
- Wrapper nodes for creating spiking reservoirs using PyNN-compatible neural network simulators (Davison et al., 2008).
- A logistic regression node trainable with different optimizers such as IRLS, conjugate gradient, BFGS and others.

1. We refer to the MDP website <http://mdp-toolkit.sourceforge.net/> for an exhaustive list.

- A Conditional Restricted Boltzmann Machine node: a standard RBM with an additional context vector.
- Several ‘utility’ signal processing nodes: a resampling node, a timeshift node, a winner-take-all node, ...

Additionally, Oger supports backpropagation training using various methods of gradient descent, such as stochastic gradient descent, RPROP and others. Finally, a FreerunFlow allows easy training and execution of architectures with feedback, e.g., for timeseries generation tasks (see the usage example below).

2.2 Validation and optimization

Around the data processing algorithms described above, which are encapsulated in nodes, Oger offers functionality for large-scale validation and optimization. The validation automates the process of constructing training and testsets, and the actual training and evaluation. Several standard validation schemes are provided (n-fold, leave-one-out (LOO) cross-validation, ...), but this can be customized (e.g. if a fixed training and test set is defined).

Oger provides an Optimizer class. This class allows both exploration of a certain parameter space and optimization of a vector of parameters according to a loss function (which can be user-defined, or one of the several provided by Oger). The optimization itself can be done using grid-searching, or using an interface to any of the algorithms in `scipy.optimize` or the Python CMA-ES module (Hansen, 2006). Finally, a variety of error measures and utility classes such as a ConfusionMatrix are included.

2.3 Parallel execution

Oger allows two modes of parallel execution, both local (multi-threaded or multi-process) and on a computing grid. The first mode is inherited from MDP, where the training and execution of a flow on a dataset consisting of different chunks can be done in parallel (if the nodes in the flow support this). The second mode is the parallel evaluation of parameter points for grid-searching and CMA-ES (the `scipy.optimize` functions as yet do not support this). Both modes use runtime overloading of class methods by their parallel versions, which makes the transition from sequential to parallel execution very user-friendly and possible using a couple of lines of code (see the usage example below).

3. Usage example

As an illustrative example, we construct and train a reservoir and readout setup with output feedback for generating the Mackey-Glass timeseries. We refer to the Oger website and the Oger installation package for more usage examples.

```

1 from scipy import *
2 import Oger, mdp
3 signals = Oger.datasets.mackey_glass(n_samples=4, sample_len=3000)
4 res = Oger.nodes.LeakyReservoirNode(output_dim=400, reset_states=False)
5 readout = Oger.nodes.RidgeRegressionNode()
6 flow = Oger.nodes.FreerunFlow([res, readout], freerun_steps=300)

```

```

7 parameters = {res: {'input_scaling': arange(.1, 1, .1), 'bias_scaling':
  arange(0, .5, .1), 'leak_rate': arange(.1, .5, .1)}}
8 internal_params = {readout: {'ridge_param': 10. ** arange(-4, 0, .5)}}
9 opt = Oger.evaluation.Optimizer(parameters, Oger.utils.nrmse)
10 opt.scheduler = mdp.parallel.ProcessScheduler(n_processes=None)
11 mdp.activate_extension('parallel')
12 opt.grid_search([[[]], signals[:-1]], flow,
  Oger.evaluation.leave_one_out, internal_params)
13 opt_flow = opt.get_optimal_flow(verbose=True)
14 opt_flow.train([[[]], signals[:-1]])
15 y = opt_flow.execute(signals[-1][0])

```

On line 3, the dataset is generated (Oger provides out of the box many benchmark tasks used in literature), which in this case consists of four Mackey-Glass timeseries generated from different initial states. In the next two lines, a reservoir node and a linear readout node trained with ridge regression are created. Line 6 concatenates these nodes into a FreerunFlow, which provides one-step ahead prediction during training and feeds the output back to the input of the flow during execution. Lines 7 and 8 define a search space for the reservoir parameters and the regularization constant of the readout node which is optimized separately for each set of reservoir parameters. On line 9 an Optimizer object is instantiated which will optimize this parameter using the provided error measure (Normalized Root Mean Squared Error). Lines 10 and 11 ensure that the optimization is done in parallel, using separate processes. On line 12, the actual optimization is performed using LOO cross-validation on the four timeseries, while for each fold the regularization constant for the ridge regression is optimized again using LOO cross-validation. On line 13 the Optimizer is queried to return the optimal flow, which is subsequently trained using all the training signals and applied to an unseen test signal in lines 14 and 15 respectively.

Acknowledgments

This work was funded by the European Commission FP7 project ORGANIC (FP7-231267).

References

- A.P. Davison, D. Brüderle, J. Eppler, J. Kremkow, E. Müller, D. Pecevski, L. Perrinet, and P. Yger. PyNN: a common interface for neuronal network simulators. *Frontiers in Neuroinformatics*, 2, 2008.
- N. Hansen. The CMA evolution strategy: a comparing review. In *Towards a new evolutionary computation*, pages 75–102. Springer, 2006.
- David Verstraeten, Benjamin Schrauwen, Michiel D’Haene, and Dirk Stroobandt. A unifying comparison of reservoir computing methods. *Neural Networks*, 20:391–403, 2007.
- 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.