Journal of Machine Learning Research 13 (2012) 2995-2998 Submitted 9/11; Revised 7/12; Published 10/12

Oger: Modular Learning Architectures For Large-Scale Sequential Processing

David Verstraeten DAVID.VERSTRAETEN@UGENT.BE
Benjamin Schrauwen BENJAMIN.SCHRAUWEN@UGENT.BE
Sander Dieleman SANDER.DIELEMAN@UGENT.BE
Philemon Brakel PHILEMON.BRAKEL@UGENT.BE
Pieter Buteneers PIETER.BUTENEERS@UGENT.BE
Department of Electronics and Information Systems
Ghent University
Ghent, Belgium

Dejan Pecevski DEJAN@IGI.TUGRAZ.AT
Institute for Theoretical Computer Science
Graz University of Technology
Graz, Austria

Editor: Cheng Soon Ong

Abstract

Oger (OrGanic Environment for Reservoir computing) is a Python toolbox for building, training and evaluating modular learning architectures on large data sets. It builds on MDP for its modularity, and adds processing of sequential data sets, 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 Oger toolbox originated from the need to rapidly implement, investigate and compare complex architectures built from state-of-the-art sequential processing algorithms, focused on but not limited to reservoir computing, and to apply these architectures to large real-world tasks. Reservoir computing (RC) is a learning framework (Verstraeten et al., 2007) whereby a random non-linear dynamical system (usually a recurrent neural network) is left untrained and used as input to a simple learning algorithm such as linear regression. A number of smaller toolboxes for reservoir computing are available, written in C++, Java and Matlab.[1] However, these are generally focused on specific implementations of RC (echo state networks or liquid state machines) and offer less flexibility in creating and evaluating complex architectures.

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 mod-

1. An overview can be found at `http://organic.elis.ugent.be/software`.

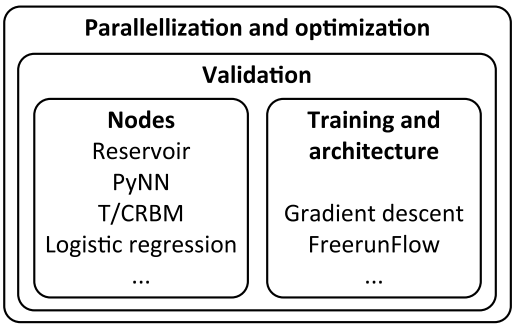

Figure 1: A schematic overview of the structure of Oger. The basic processing blocks (nodes) are combined with methods for constructing and training architectures. These architectures can then be evaluated in a validation and optimization framework.

ularity was one of the key requirements for Oger, it has been based on the well known and widely used Modular Data Processing toolkit (MDP), which provides this modularity in addition to a wide variety of machine learning algorithms (Zito et al., 2008). 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 type-checking is abstracted away through the object-oriented interface, such that the developer can focus on implementing the actual algorithm.

Python was chosen as the development language because it is a high-level, cross-platform and open-source interpreted language offering flexibility and rapid development, while interfaces to optimized numerical linear algebra packages such as BLAS 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.

2. Features

In this section we describe the main features of Oger and give a usage example.

2.1 Algorithms

MDP implements several standard supervised and unsupervised learning methods for operating on stationary inputs, such as principal component analysis, independent component analysis and factor analysis.[2] Oger adds several new methods to this set:

– Several reservoir implementations : a basic reservoir with customizable nonlinear function and weight topologies, a leaky integrator reservoir, and a GPU-optimized reservoir using CUDA.

– Wrappers 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.

2. We refer to the MDP website `http://mdp-toolkit.sourceforge.net/` for an exhaustive list.

– A conditional restricted Boltzmann machine: a standard RBM with an additional context vector.

– Several 'utility' signal processing methods: a resampling node, a timeshift node, a winner-take-all node, and others.

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, for instance for time-series generation tasks (see the usage example below).

2.2 Validation, Optimization and Parallel Execution

Around the data processing algorithms described above, Oger offers functionality for large-scale validation and optimization. The validation automates the process of constructing training and test sets, and the actual training and evaluation. Several standard validation schemes are provided (n-fold, leave-one-out (LOO) cross-validation and others), but this can be customized (for example, 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.

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 data set 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 time-series. 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 data set is generated, which in this case consists of four Mackey-Glass time-series 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 these parameters 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 time-series, while for each fold the regularization constant for the ridge regression is optimized again using LOO cross-validation. This can take a few minutes. 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). Dejan Pecevski has been additionally partially supported by the European Union project FP7-506778 (PASCAL2).

References

A.P. Davison, D. Brüderle, J. Eppler, J. Kremkow, E. Muller, 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.

D. Verstraeten, B. Schrauwen, M. D'Haene, and D. 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.