# Oger toolbox

## About the toolbox


The OrGanic Environment for Reservoir computing (Oger) toolbox is a Python toolbox, released under the LGPL, for rapidly building, training and evaluating modular learning architectures on large datasets. It builds functionality on top of the Modular toolkit for Data Processing (MDP). Using MDP, Oger provides:

- Easily building, training and using modular structures of learning algorithms
- A wide variety of state-of-the-art machine learning methods, such as PCA, ICA, SFA, RBMs, ... You can find the full list here.

The Oger toolbox builds functionality on top of MDP, such as:

- Cross-validation of datasets
- Grid-searching large parameter spaces
- Processing of temporal datasets
- Gradient-based training of deep learning architectures
- Interface to the Speech Processing, Recognition, and Automatic Annotation Kit (SPRAAK)

In addition, several additional MDP nodes are provided by Oger, such as a:

- Reservoir node
- Leaky reservoir node
- Ridge regression node
- Conditional Restricted Boltzmann Machine (CRBM) node
- Perceptron node

## Installation

See here for instructions on downloading and installing the toolbox.

## Getting started

There is a general tutorial and examples highlighting some key functions of Ogerhere. A pdf version of the tutorial pages is here.

## API documentation

You can find an automatically generated API documentationhere.

## Bugs and feature requests

You can submit bugs or requests for additional features using the issue tracking system at github.ugent.be for this repository.

# Examples

In the /examples directory of the Oger package, there are a number of simple scripts which show how to run some simple standard benchmark tests using a number of architectures and training methods. Some of these examples are also explained step by step in the links below.

# 30th order NARMA tutorial

**A 30th order Non-linear AutoRegressive Moving Average (NARMA) task with a standard reservoir**

This code is available as examples/narma30_demo.py.

We first create the dataset, with a sample length of 1000 timesteps, and ten examples (this is the default value, so it is not explicitly passed) in total.

```
[x, y] = Oger.datasets.narma30(sample_len=1000)
```

Both x and y are lists of 2D numpy arrays. Oger inherets the dimensionality convention from MDP, so the rows represent timesteps and the columns represent different signals.

We then construct the reservoir node:

```
reservoir = Oger.nodes.ReservoirNode(input_dim=1, output_dim=100,
input_scaling=0.05)
```

As you can see, we specify a number of parameters of the reservoir by passing keyword arguments, such as the input-dimensionality, output dimensionality and the scaling of the input weights. You can find a full list of the ReservoirNode arguments and their default values in the documentation (TODO: link).

Next, we construct the readout node which is a linear node trained using ridge regression.

```
readout = Oger.nodes.RidgeRegressionNode()
```

We use the two nodes to construct a flow and turn on verbosity:

```
flow = mdp.Flow([reservoir, readout], verbose=1)
```

We take the first nine samples of x and y, and put them in a list as follows:

```
data = [x[0:-1], zip(x[0:-1], y[0:-1])]
```

This is because flows expect their data in a certain format (taken from the MDP documentation):

*'data_iterables' is a list of iterables, one for each node in the flow.The iterators returned by the iterables must return data arrays that are then used for the node training (so the data arrays are the 'x' for the nodes). Note that the data arrays are processed by the nodes which are in front of the node that gets trained, so the data dimension must match the input dimension of the first node.*
*Instead of a data array 'x' the iterators can also return a list or tuple, where the first entry is 'x' and the following are args for the training of the node (e.g. for supervised training).*

So, in our case, the first element of the list (the x[0:-1]) is used as input for the reservoir node, and the second element of the list (the zip(x[0:-1], y[0:-1])) is used to train the linear readout, by feeding the first argument of the zip() as input to the first node, and using the second argument of the zip() to as target for training.

Our flow can then be trained on the training data:

```
flow.train(data)
```

Applying the trained flow to test-data (the tenth example in the dataset, remember that Python uses zero-based indexing!) is then as simple as:

```
testout = flow(x[9])
```

We can then compute and print the NRMSE as follows:

Architecture of the hierarchy:

```
print "NRMSE: " + str(Oger.utils.nrmse(y[9], testout))
```

# Classification of MNIST data using a Deep-Belief Network with perceptron readout

This code is available as examples/DBN_classifier_demo.py

In this example a deep belief network is constructed and trained on the MNIST handwritten digits dataset. The deep belief network consists of two layers of Restricted Boltzmann Machines (RBM) and a logistic regression classification layer. This example uses the new ERBMNode that has been implemented in the toolbox. The logistic regression is implemented in the form of a perceptron node with softmax outputs.

```
rbmnode1 = Oger.nodes.ERBMNode(784, 100) rbmnode2 = Oger.nodes.ERBMNode(100, 200)
percnode = Oger.nodes.PerceptronNode(200, 10,
transfer_func=Oger.utils.SoftmaxFunction)
```

The two RBM layers are first trained in an unsupervised way. The first RBM is trained to reconstruct the input data. The second RBM is trained to reconstruct the output of the first RBM.

```
for epoch in range(epochs): for c in mdp.utils.progressinfo(train_data):
rbmnode1.train(c.reshape((1, 784)), n_updates=1, epsilon=.1) hiddens =
rbmnode1(train_data) for epoch in range(epochs): for c in
mdp.utils.progressinfo(hiddens): rbmnode2.train(c.reshape((1, 100)), n_updates=1,
epsilon=.1)
```

Subsequently, a flow is created that stacks all the layers on top of each other and passed to a backprop node for gradient descent training.

```
myflow = rbmnode1 + rbmnode2 + percnode bpnode = Oger.gradient.BackpropNode(myflow,
Oger.gradient.GradientDescentTrainer(momentum=.9), loss_func=Oger.utils.ce)
```

This backprop node is used to fine-tune the network for classification. After testing, the proportion of correctly classified images is reported and the input weights of the first layer before and after fine-tuning are displayed as images.

Note that this demo uses far less units than in a typical deep belief network experiment and that only a single epoch of stochastic gradient descent is carried out.

# Constructing a Temporal Reservoir Machine hierarchy

The code for this example can be found at examples/trm_hierarchy_demo.py

In this example, a hierarchy of conditional Restricted Boltzmann machines (or Temporal Reservoir Machines) is constructed and trained on some trivial toy task. Between every layer, Principal Component Analysis (PCA) is applied to the data that is fed to the reservoir in the next layer. It demonstrates how the MDP/Oger functionality can be used to construct complex hierarchical models for temporal data.

As shown below, the architecture consists of six layers. The first layer contains both a ShiftNode that shifts data to the left or to the right over time and a ReservoirNode. The output of this layer is concatenated and fed to the CRBMNode that uses the reservoir's output as context data to model the data that arrives through the shift node. The activation patterns of the hidden units of the CRBM are fed to both another ShiftNode and a PCA node. In the next layer, the Identity node receives the shifted data while the reservoir receives the PCA compressed data. The output of these two nodes is again fed to a CRBM layer in the same way as before. Finally, the activation patterns of the hidden units of this second CRBM are fed to a node that performs ridgeregression.

<table>
<tr><td></td><td>ShiftNode</td><td>ReservoirNode</td></tr>
<tr><td colspan="3">CRBMNode</td></tr>
<tr><td>ShiftNode</td><td>PCANode</td><td></td></tr>
<tr><td>IdentityNode</td><td>ReservoirNode</td><td></td></tr>
<tr><td colspan="3">CRBMNode</td></tr>
<tr><td colspan="3">RidgeRegressionNode</td></tr>
</table>

Constructing and training this architecture is done using the following steps:

First, the components of the first layer are constructed and combined in a layer that makes sure they receive the same input.

```
reservoir1 = Oger.nodes.ReservoirNode(input_dim=20, output_dim=300) shift1 = Oger.nodes.ShiftNode(input_dim=20, n_shifts=1) ReservoirLayer1 = mdp.hinet.SameInputLayer([shift1, reservoir1])
```

Second, the data is passed through this first layer, a CRBM is trained on it and added to the architecture's flow.

```
x = ReservoirLayer1(u) crbmnode1 = Oger.nodes.CRBMNode(hidden_dim=crbm1_size, visible_dim=20, context_dim=300) for epoch in range(epochs): for i in range(len(x) - 1): crbmnode1.train(x[i:i + 1, :], epsilon=.001, decay=.0002) theflow = ReservoirLayer1 + crbmnode1
```

The next layer is again a SameInputLayer but now contains a shift node and a PCA node. The layer with a reservoir that follows the PCA layer, is constructed using a normal hinet layer so the input of the IdentityNode and the reservoir is not the same.

```
ReservoirLayer2 = mdp.hinet.Layer([identity, reservoir2])
```

A second CRBM is trained on the output of this last layer and added to the architecture's flow as well. Finally, a RidgeRegression node is trained and added to the flow.

```
readout = Oger.nodes.RidgeRegressionNode(ridge_param=.0000001, input_dim=crbm2_size, output_dim=20) readout.train(x, t) readout.stop_training() theflow += readout
```

Passing data (in this case the variable 'u') through the architecture to obtain predictions is now very simple.

```
theflow = mdp.hinet.FlowNode(theflow)
```

```
y = theflow(u)
```

# Cross-validation tutorial

## This code is available as examples/cross_validation_demo.py.

N-fold cross-validation involves splitting the data into N chunks, where N is the number of folds. For each fold, all-but-one of the chunks is used for training and the remaining chunk for testing. This is repeated so that each chunk is used exactly once for testing. The cross-validation error is then the mean error over all cross-validation folds. In the special case where the number of folds is equal to the number of samples, this is called leave-one-out crossvalidation.

Oger supports cross-validation of datasets. As a tutorial example, we will redo the 30th order NARMA experiment, but this time using different forms of cross-validation. We start of with the same dataset and flow definitions:

```
inputs, outputs = Oger.datasets.narma30(sample_len=1000) data = [inputs, zip(inputs, outputs)] reservoir = Oger.nodes.ReservoirNode(output_dim=100, input_scaling=0.1) readout = Oger.nodes.RidgeRegressionNode(0) flow = mdp.Flow([reservoir, readout])
```

Next, instead of training and executing the flow manually on training and testing data, we use the Oger function validate:

```
errors = Oger.evaluation.validate(data, flow, Oger.utils.nrmse, cross_validate_function=Oger.evaluation.train_test_only, training_fraction=0.5)
```

This function takes a dataset (in the MDP format, i.e. a list of iterables) and a flow, and additionally a loss-function (in this case the nrmse), a cross-validation function and additional keyword-arguments which are passed on to the cross-validation function (in this case training_fraction=0.5).

The cross-validation function determines the type of cross-validation: here we use the special case of simple training and testing, using 9/10 of the dataset for training (training_fraction=0.9). So, this is equivalent to the original NARMA tutorial except that here the examples used for training are selected randomly. The validate function returns a list of errors, one for each fold (which is in this case only a single value).

We can repeat this same experiment but using 5-fold cross-validation as follows:

```
errors = Oger.evaluation.validate(data, flow, Oger.utils.nrmse, n_folds=5)
```

Notice how we don't explicitly pass the cross-validation-function, because we want the default value of Oger.evaluation.n_fold_random. We also use a different keyword argument for the cross-validation function, namely the number of folds. The return argument errors is in this case a list of five values.

Finally, we can maximize the amount of data used for training by using leave-one-out cross-validation as follows:

```
errors = Oger.evaluation.validate(data, flow, Oger.utils.nrmse, cross_validate_function=Oger.evaluation.leave_one_out)
```

This cross-validation function does not use any additional keyword arguments, since the number of folds is determined by the size of the dataset (in

this case 10 examples). The errors will be a list of 10 values in this case.

# Gradient-based training of a flow

This code is available as examples/gradient_demo.py

The demo demonstrates gradient based learning by training a two layer feedforward neural network to learn a specific non-linear mapping using the gradient package.

In this case the sine function is used. For training, we use random samples from this function.

```
data_in = np.random.uniform(-1, 1, 50) data_out = np.sin(data_in * 5 * np.pi)
```

After this, a flow of two perceptron nodes is created.

```
percnode1 = Oger.nodes.PerceptronNode(1, 12, transfer_func=Oger.utils.TanhFunction)
percnode2 = Oger.nodes.PerceptronNode(12, 1) myflow = percnode1 + percnode2
```

The user is asked to select an optimization algorithm. Several of these so-called trainers have been defined already and they include gradient descent, RPROP, conjugate gradient and BFGS.

The chosen optimization algorithm is passed as an argument during the construction of a backprop node which is trained on the data.

```
bpnode = Oger.gradient.BackpropNode(myflow, choices[choice],
loss_func=Oger.utils.mse) bpnode.train(x=data_in, t=data_out)
```

The results are displayed as a plot that contains the original sine function, blue dots that indicate the training data that was available to the network and the function the network has learned.

# Grid-searching and plotting

This code is available as examples/plotting_example.py

If you want to sweep one or more parameters of a flow on a dataset using cross-validation, you can use the Oger.evaluation.Optimizer class for this. We will show how you can use an object of this class to gridsearch a parameter space and plot the results afterwards.

We construct the dataset (30th order NARMA) and the flow in the standard way (see the NARMA tutorial for more information on this).

```
[inputs, outputs] = Oger.datasets.narma30() data = [inputs, zip(inputs, outputs)]
reservoir = Oger.nodes.ReservoirNode(input_scaling=0.1, output_dim=100) readout =
Oger.nodes.RidgeRegressionNode(0.001) flow = mdp.Flow([reservoir, readout])
```

Next, we define the parameters and ranges we want to sweep:

```
gridsearch_parameters = {reservoir:{'_instance':range(5),
'spectral_radius':mdp.numx.arange(0.6, 1.3, 0.1)}}
```

This variable is a dictionary of dictionaries. In the top-level dictionary, the keys are node instances (just one here: reservoir). The corresponding value is again a dictionary, where every key is a parameter (given as a string), and the dictionary value is a range of values the parameter should take. So, in the example above, we range for the reservoir node over the _instance parameter with values 0 to 4, and over the spectral radius with values from 0.6 to 1.3 in steps of 0.1.

The _instance parameter is a dummy parameter, which allows you to instantiate different reservoirs for the same parameter settings. These results can then be averaged (see later in this tutorial).

We can now instantiate an optimizer, passing it the parameter dictionary defined above and the desired loss function:

```
opt1D = Oger.evaluation.Optimizer(gridsearch_parameters, Oger.utils.nrmse)
```

We can then do the actual grid-search as follows:

```
opt1D.grid_search(data, flow, n_folds=3,
cross_validate_function=Oger.evaluation.n_fold_random)
```

The keyword arguments cross_validate_function and any additional keyword arguments are passed on internally to the Oger.evaluation.validate() function - see the cross-validation tutorial for more info on this.

Once the grid-search is done, we can plot the results as follows:

```
opt1D.plot_results([(reservoir, '_instance')])
```

The argument being passed is a list of (node, parameter_string) tuples. This argument is optional, but if it is passed, first the optimizer takes the mean over the given parameters and then does the plotting. If only one element is given (as is the case here), and the resulting plot is 1D (i.e. one parameter remains), then errorbars are plotted showing the variance w.r.t. the parameter being averaged over. So in this case, the errorbars show the variance over different reservoir instantiations.

The plot_results function automatically detects if a 1D or 2D plot is necessary. Let's try a 2D gridsearch with again several reservoir instances per parameter setting. We instantiate a new optimizer with a different gridsearch dictionary and run the gridsearch:

```
gridsearch_parameters = {reservoir:{'spectral_radius':mdp.numx.arange(0.6, 1.3,
0.2),'input_scaling': mdp.numx.arange(0.5, .8, 0.1),'_instance':range(5)}} opt2D =
Oger.evaluation.Optimizer(gridsearch_parameters, Oger.utils.nrmse) errors =
opt2D.grid_search(data, flow, n_folds=3,
cross_validate_function=Oger.evaluation.n_fold_random)
opt2D.plot_results([(reservoir, instance)])
```

We call the plot_results function, again taking the mean over reservoir instances.

This then gives in a 2D image plot.

# Isolated spoken digit recognition using a leaky integrator reservoir

This code is available as examples/analog_speech.py.

In this example, we will use a reservoir consisting of leaky integrator neurons to perform classification of isolated spoken digits (zero to nine, i.e. ten classes). The speech was already pre-processed using a cochlear model by R.F. Lyon, which results in a 77-dimensional set of input signals per digit.

We start by constructing the dataset:

```
[inputs, outputs] =
Oger.datasets.analog_speech(indir="/afs/elis/group/snn/speech_corpora/ti46_subset/Lyon_de
```

Here, outputs is a ten-dimensional signal, with +1 encoding the current digit class and -1 for all the remaining classes.

Next, we construct the nodes and create a flow:

```
input_dim = inputs[0].shape[1] reservoir =
Oger.nodes.LeakyReservoirNode(input_dim=input_dim, output_dim=100, input_scaling=1,
leak_rate=0.1) readout = Oger.nodes.RidgeRegressionNode(0.001) mnnode =
Oger.nodes.MeanAcrossTimeNode() flow = mdp.Flow([reservoir, readout, mnnode])
```

Note how we pass a leak_rate=0.1 argument to the constructor. The argument passed to the ridge regression node is the regularization parameter, to ensure good generalization on the test data. This should really be optimized per reservoir and dataset, but to reduce computation time we pre-specify it here (so the result will be sub-optimal).

We also use an additional utility node, the MeanAcrossTimeNode, which takes the mean value over time of each of its input signals. This is needed because we need a single output vector for classification purposes.

We then determine how many samples to use for training and testing, and train the flow:

```
train_frac = .9 n_samples = len(inputs) n_train_samples = int(round(n_samples *
train_frac)) n_test_samples = int(round(n_samples * (1 - train_frac)))
flow.train([None, \ zip(inputs[0:n_train_samples - 1], \ outputs[0:n_train_samples -
1]), \ [None]])
```

Note how we pass a list of three iterables now, one for each node in the flow.

Finally, we apply the flow to the test data and compute the test error:

```
for xtest in inputs[n_train_samples:]: ytest.append(flow(xtest)) ytestmean = sp.array([sp.argmax(sample) for sample in ytest])
```

We can now the ConfusionMatrix class to compute several metrics and error measures that are typically used in classification problems.

First, we construct the 10-class confusion matrix using the output of the system and the desired output labels.

```
confusion_matrix = ConfusionMatrix.from_data(10, ytestmean, ymean) # 10 classes
```

Many commonly used error measures and metrics can now be obtained as properties of the ConfusionMatrix:

```
print "Error rate: %.4f" % confusion_matrix.error_rate # this comes down to 0-1 loss
print "Balanced error rate: %.4f" % confusion_matrix.ber
```

We can also reduce the 10-class problem to 10 binary classification problems, where each class in turn is chosen to be the positive class, and all the others are grouped into the negative class. The 'binary' method of the ConfusionMatrix class generates the corresponding binary confusion matrices. Binary confusion matrices have some additional methods to compute measures like precision, recall, etc. which do not generalise to multiclass-classification problems.

```
print "Per-class precision and recall" binary_confusion_matrices = confusion_matrix.binary() for c in range(10): m = binary_confusion_matrices[c] print "label %d - precision: %.2f, recall %.2f" % (c, m.precision, m.recall)
```

It is also possible to create functions that compute error measures based on confusion matrices using the 'error_measure' method of the ConfusionMatrix class. This method returns a function that takes observed outputs and desired outputs as input, and computes the desired error measure.

```
ber = ConfusionMatrix.error_measure('ber', 10) # 10-class balanced error rate function print "Balanced error rate: %.4f" % ber(ytestmean, ymean)
```

Finally, we can visualise the confusion matrix using the plot_conf function. In most cases it is recommended to balance (row-normalise) a confusion matrix before visualisation, particularly when the number of training samples per class is not the same for every class. The ConfusionMatrix class has a 'balance' (or 'normalise_per_class') method for this purpose.

```
plot_conf(confusion_matrix.balance())
```

# Learning an artificial grammar using a reservoir

The example code can be found at examples/grammar_task.py

In this example, an Echo-State Network is trained to do next-word prediction on a set of sentences that have been constructed by using a probabilistic context free grammar.

An example grammar from the datasets module is used to generate training sentences.

```
l = Oger.datasets.simple_pcfg() trainsents = [l.S() for i in range(N)]
```

A set of test sentences is constructed as well but since these don't need to have the correct statistical properties (they just need to be grammatical), they are created using a couple of for loops. After this, a reservoir with a linear read-out (or ESN) is created and wrapped in a flownode.

```
reservoir = Oger.nodes.ReservoirNode(inputs, 100, input_scaling=1) readout = Oger.nodes.RidgeRegressionNode() flow = mdp.Flow([reservoir, readout], verbose=1) flownode = mdp.hinet.FlowNode(flow)
```

The words of the train and test sentences are coded into vectors. All the elements of these vectors are zero, except for a single 1 that codes what specific word the vector represents. The desired output is simply the training data shifted one step in time so the network will learn to predict the next word given the previous part of the sentence. The training is done using the flownode.

```
flownode.train(x, y) flownode.stop_training()
```

The test data is presented and the predictions of the network are stored.

```
ytest = flownode(x)
```

Since the network can never know with 100% certainty what the next word is going to be, performance is evaluated by taking the vector cosine between the true probability distribution over the next possible words (as calculated by a probabilistic incremental parser that knows the grammar) and the network outputs. Perferct performance would yield a cosine of 1 for every word. The scores are constrained to lie in the interval [-1, 1] but in practice they rarely go below zero.

# Mackey-Glass attractor - signal generation

This code is available as examples/signal_generation.py

In this task, we will train a standard reservoir + readout to do autonomous signal generation. This is done by training the readout to perform one-step-ahead prediction on a teacher signal. After training, we then feed the reservoir its own prediction, and it then autonomously generates the signal for a certain time.

MDP only supports feed-forward flows, so how can we solve this? Oger provides a FeedbackFlow which is suited for precisely this case. The training and execute functions are overridden from the original Flow class. A FeedbackFlow is suitable for signal generation tasks both without and with external additional inputs. The example below uses no external inputs, so the flow generates its own output. After training, for every timestep, the output of the Flow is fed back as input again, such that it can autonomously generate the desired signal.

The train function of a FeedbackFlow will take the timeseries given as training argument and internally construct a new target signal which is shifted one timestep into the past, such that the flow is trained to do one step ahead prediction. During execution (i.e. by calling the execute function), the flow will first be teacher-forced using the first portion provided input signal, and starting from the timestep freerun_steps from the end of the signal, the flow is run in freerun mode, i.e. being fed back its own prediction.

We start by preparing the dataset. In this case, there is no target signal, so the dataset consists of only a single time-series. We construct the target signal as the time-series shifted over one timestep to the left (we want one-step ahead prediction).

```
freerun_steps = 1000 training_sample_length = 5000 n_training_samples = 3
test_sample_length = 5000 train_signals =
Oger.datasets.mackey_glass(sample_len=training_sample_length,
n_samples=n_training_samples)
```

Next, we create a reservoir with a little bit of leak rate and a readout node and combine these into a FeedbackFlow. Notice how during creation of the FeedbackFlow, we pass it the keyword argument of freerun_steps. This determines how many timesteps the FeedbackFlow will run in 'freerun mode'.

```
reservoir = Oger.nodes.LeakyReservoirNode(output_dim=400, leak_rate=0.8,
input_scaling=.4, bias_scaling=.2, reset_states=False) readout =
Oger.nodes.RidgeRegressionNode() flow = Oger.nodes.FreerunFlow([reservoir, readout],
freerun_steps=freerun_steps)
```

We instantiate an Optimizer for finding the best regularization constant for the readout. This is necessary to achieve a suitable robustness and stability of the generated model.

```
gridsearch_parameters = {readout:{'ridge_param': 10 ** scipy.arange(-4, 0, .3)}}
loss_function = Oger.utils.timeslice(range(training_sample_length - freerun_steps,
training_sample_length), Oger.utils.nrmse) opt =
Oger.evaluation.Optimizer(gridsearch_parameters, loss_function)
opt.grid_search([[], train_signals], flow,
cross_validate_function=Oger.evaluation.leave_one_out)
```

We run a gridsearch of the ridge parameter, providing the set of training signals as dataset, and using leave one out cross-validation.

We then retrieve the optimal flow, train it on the full training set and evaluate its performance on the unseen test signals.

```
opt_flow = opt.get_optimal_flow(verbose=True) opt_flow.train([[], train_signals])
freerun_output = opt_flow.execute(test_signals[0][0])
```

To check the results, we can plot an overlay of the signal generated by the reservoir, and the actual target signal:

```
pylab.plot(scipy.concatenate((test_signals[0][0][-2 * freerun_steps:])))
pylab.plot(scipy.concatenate((freerun_output[-2 * freerun_steps:])))
```

# Modelling of place-cells for a robot with a reservoir + SFA and ICA

This code is available as examples/place_cells_sfa.py

The following implements an example of the experiments run in [1], by E. Antonelo and B. Schrauwen.

It takes sensor data recorded during a robot's run through an environment. It feeds this data through a hierarchy of a reservoir, then slow feature analysis, then independent component analysis. It can be seen that certain outputs repeatedly produce a spike of activity at precise locations. This is learned in an unsupervised manner.

The different data sets all are within one matlab file. This contains: 'data_info', which holds xy position, location number, etc. 'sensors' which has the sensor recordings at each time step. And finally 'sensors_resampled', a downsampling of the sensor data with x50 less timesteps. These can be extracted in to python using the 'loadmat' function in Scipy.

The example first invoves defining the reservoir. In the paper a reservoir size of 400, leak rate of 0.6 and spectral radius of 0.9 were used. The input dimension is taken from the number of dimensions in the sensor data. Also, the input weight matrix (set with resNode.w_in) had values of -0.2,0.2 and 0 with probability of 0.15, 0.15 and 0.7 respectively.

```
resNode = Oger.nodes.LeakyReservoirNode(input_dim=sensorData.shape[0], output_dim=400, spec_radius=0.9, leak_rate=0.6) resNode.w_in = w_in resNode.initialize()
```

The SFA and ICA layers both contain 70 outputs. They are added to the reservoir node in an MDP Flow node. This flow node is then defined, trained and executed on the sensor data.

```
sfaNode = mdp.nodes.SFANode(output_dim=70) icaNode = mdp.nodes.FastICANode() icaNode.set_output_dim(70) flow = mdp.Flow(resNode + sfaNode + icaNode) flow.train(sensorData.T) icaOutput = flow.execute(sensorData.T)
```

Depending on which of the datasets was used, the icaOutput may need to be resampled to be properly compared with the location and XY robot data. This can be easily done with the resample method in scipy.

```
icaOutputLong = resample(icaOutput, newNumberOfSamples)
```

We can then plot some (below given value of 20) of the ICA outputs against the location number of the robot.

```
plotsNum = 20 plt.subplot(plotsNum+1, 1, 1) # first subplot of the numbered location of the robot plt.plot(location) # plot the independent components for i in range(plotsNum): plt.subplot(plotsNum+1, 1, i+2) plt.plot(icaOutputLong[:,i]) plt.show()
```

**References**


- Antonelo, E. A., B. Schrauwen, and D. Stroobandt, "**Unsupervised Learning in Reservoir Computing: Modeling Hippocampal Place Cells for Small Mobile Robots**", *International Conference on Artificial Neural Networks (ICANN)*, 2009.


# Parallelization on a single machine or a computer cluster

Oger uses theParallel Python package for parallelization. This light-weight package uses a worker-server architecture to distribute jobs, and can cope with both SMP (on one machine) as cluster (multiple machines) parallelization.

It is possible to parallelize computations using Oger in two ways:

- dividing the dataset into chunks, where the simulation of each chunk is done by a separate worker, and the results are recombined afterwards.
- running a grid-search in parallel by simulating every parameter point on a separate worker.

## Parallelization of datasets

The first method of parallelization is illustrated in the script examples/narma30_demo_parallel.py. This is actually a parallel version of the narma30_demo.py example. The main difference lies in the creation of a ParallelFlow instead of a standard Flow:

```
flow = mdp.parallel.ParallelFlow([reservoir, readout], verbose=1)
```

We also need a scheduler to handle the distribution of the jobs. MDP provides two standard schedulers for parallel execution on a single machine: one thread-based and one process-based. The thread-based one is not suitable for running a reservoir, because it uses the same objects in different threads, and these would overwrite each other's states. Stateless nodes however are fine to use thread-based. In this case we want to execute the jobs on a single multicore machine using separate Python processes, so we use the ProcessScheduler provided by MDP:

```
scheduler = mdp.parallel.ProcessScheduler(n_processes=2, verbose=True)
```

This scheduler is passed when training the flow:

```
flow.train(data, scheduler)
```

We need to shut down the scheduler afterwards:

```
scheduler.shutdown()
```

## Parallelization of parameter sweeps

It is also possible to parallelize a gridsearch of a parameter sweep by executing the evaluation of the flow in each parameter point as a separate job. This is illustrated in the script examples/gridsearch_parallel.py. Again, the difference with the unparallelized version is minimal:

We first create a dictionary with the gridsearch parameters and their corresponding ranges:

```
gridsearch_parameters = {reservoir:{'input_scaling': mdp.numx.arange(0.1, 1, 0.2),
'spectral_radius':mdp.numx.arange(0.1, 1.5, 0.3)}}
```

We use this dictionary to instantiate an Optimizer object:

```
opt = Oger.evaluation.Optimizer(gridsearch_parameters, Oger.utils.nrmse)
```

A process-based scheduler (for local execution using multiple processes) is added to the optimizer:

```
opt.scheduler = mdp.parallel.ProcessScheduler(n_processes=2, verbose=True)
```

We then use the extension mechanism of MDP to activate the parallel execution of the gridsearch. This extension mechanism internally replaces the standard serial grid-search method of the Optimizer object with a parallel version.

```
mdp.activate_extension("parallel") opt.grid_search(data, flow,
cross_validate_function=Oger.evaluation.n_fold_random, n_folds=5)
```

Once the execution is done we can query the optimizer for the minimal error:

```
min_error, parameters = opt.get_minimal_error()
```

## Parallelization on a computer cluster

#### The single user case

The examples given above can equally well be run on a computer cluster. This is done by supplying a Parallel Python server instead of a ThreadScheduler to the parallelization mechanism:

```
job_server = pp.Server(0, ppservers=("node1:60000",?node2:60000?, ?node3:60000?))
```

Here, the ppservers keyword argument should be a tuple of hostname:port strings, simply hostnames if the default Parallel Python port is used.

Before executing the script, the nodes on the cluster should be running a ppserver.py process.

#### The multiple user case

In case multiple users have access to the computer cluster, this will probably require some kind of resource and user management mechanism. There are some very powerful and widely used platforms for this such as Torque, Sun Grid Engine or Condor. Regardless of the actual resource manager which is used, the basic principle remains the same: one submits as many jobs to the resource manager as there are parallel jobs to be simulated, and every job simply consists of starting a ppserver.py instance. Parallel Python has the option of passing a so-called secret key which can be used to couple ppserver instances running on the workers to the user and batch job at hand. Care should be taken that the ppserver processes shut down after job execution to free the nodes when necessary.

# Oger tutorial

## Overview

Oger is a Python toolbox for rapidly building, training and evaluating hierarchical learning architectures on large datasets. It builds functionality on top of theModular toolkit for Data Processing (MDP). Please read the MDP tutorial before continuing here.

In MDP, the central concept is known as a node, which is an elementary machine-learning or signal processing block. MDP includes a wide variety of algorithms out of the box (see the full list here). Nodes can be trainable - both unsupervisedly (e.g. clustering) and supervisedly (e.g. classification or regression) - or not (e.g. filters, preprocessing). These nodes can then be used to construct graph-like architectures by concatenating them into what is called a flow.

Oger packages many machine learning algorithms, but the main focus is around sequence processing algorithms. Particularly, many methods from the field of Reservoir Computing are available in the toolbox. Reservoir computing is a general computational framework, whereby a non-linear dynamical network of nodes (such as a recurrent neural network) called the reservoir is randomly created and left untrained. The response of the reservoir to the input is then used to train a simple algorithm (usually a linear method) to produce the desired output.

## Getting started

Using Oger in your code is as simple as:

```
import Oger
```

This loads all methods and Oger nodes into the namespace Oger. The functions and classes further split into these subpackages:

- Oger.datasets: several common benchmark datasets.
- Oger.evaluation: evaluation and optimization of flows.
- Oger.nodes: learning algorithms and signal processing nodes.
- Oger.gradient: gradient descent training.
- Oger.parallel: parallellization.
- Oger.utils: utility functions and classes.

The usual experiment consists of generating the dataset, constructing your flow (learning architecture) by concatenating nodes into a feedforward graph-like structure, and then simply training and appliying it or performing some optimization or parameter sweeps.

## A simple experiment

You can create a node by simpy instantiating it. For example:

```
resnode = Oger.nodes.ReservoirNode(output_dim = 100)
```

This creates a reservoir node of 100 neurons. Let's create a ridge regression readout node:

```
readoutnode = Oger.nodes.RidgeRegressionNode()
```

Flows can be easily created from nodes as follows:

```
flow = resnode + readoutnode
```

For more examples on how to construct flows, including more complex architectures, please see theMDP tutorial. Next, we can create data from one of the built-in datasets, a 30th order nonlinear autoregressive moving average system (NARMA). The input to the system is uniform white noise, the output is given by the NARMA system.

```
x,y = Oger.datasets.narma30()
```

By default, this returns two lists of ten 1D timeseries, the input and corresponding output, of 1000 timesteps each. Please see the API documentation for the arguments to this and other functions.

Flows are trained in a feedforward way, node by node starting from the front. In our case, the first node (the reservoir) is not trainable, so we don't need to provide data for that. To train the second node (the readout), we provide a list of input-output tuples, conveniently generated by the zip function. We will train on the first nine timeseries, and keep the last one separate for testing later on. This gives the following dataset usable for training the flow:

```
data = [None, zip(x[0:-1],y[0:-1])]
```

We can now train the flow to reproduce the output given the input like so:

```
flow.train(data)
```

We can now see how our trained architecture performs on unseen data:

```
plot(flow(x[-1])) plot(y[-1])
```

You should see a more or less OK match, but still not quite good. This is because we took the default parameters for the reservoir. The scaling of the input weights (given by the argument input_scaling) is an important parameter, let's see if we can optimize this. We define the range over which we want to scan this parameter as follows:

```
gridsearch_parameters = {resnode:{'input_scaling': mdp.numx.arange(0.1, 0.5, 0.1)}}
```

This is a dictionary, where the keys are nodes in the flow, and the values are again dictionaries. These internal dictionaries have parameter names as keys, and iterables as values. So in this case, we want to scan the input_scaling parameter of our resnode over the values .1 to .5 in steps of .1.

Let's try to optimize them using the Optimizer class. When creating an optimizer, we need to tell it what parameter space we want to explore (as defined above), as well as the loss function. For this problem the normalized RMSE is suitable. This gives:

```
opt = Oger.evaluation.Optimizer(gridsearch_parameters, Oger.utils.nrmse)
```

We can now optimize this flow on the given dataset, using 5-fold cross-validation and a brute-force gridsearch of the (small) parameter space.

```
opt.grid_search(data, flow, cross_validate_function=Oger.evaluation.n_fold_random,
n_folds=5)
```