



Using the EPOL Library. Example GA

EPOL is a C++ library intended to ease the use of evolutionary algorithms (EAs) for global optimisation and also for training neural networks (NNs). The library is readily offered to the community to investigate in these areas, especially in the topics including local search for training NNs.

In order to show an example of use of the EPOL library, this document explains the steps to configure and run a simple genetic algorithm (GA) to train a NN (i.e. to find the set of NN weights that make it to learn a -supervised learning- data set). The NN is faced to a 838 binary decoder problem learning task. The example NN is a feedforward NN with 3 layers, having 8-3-8 neurons, respectively.

Before getting started, we are going to describe the files contained in the source code file we provide you: `epol_source_code.zip`. Two main directories are included into the zip file: `seq_model` and `distr_model`. In the next sections we discuss the main details.

But let us take a first glance to the definition of the problem. First of all, it is recommended to take a look at the file `OCHO383_TRAIN.net`. This file contains the problem patterns definition for using during the training stage of the NN. It is dimensioned with an 80% of total set of available patterns for the problem. The rest (20%) is included in a test pattern file `OCHO383_TEST.net`.

File OCHO383_TRAIN.net

```
[NLAYERS]
3

[NEURONS_BY_LAYER]
8
3
8

[ACTIVATION_FUNCTION]
binary
0 1

[NUMBER_OF_PATTERNS]
6

[INPUT_PATTERNS]
1 0 0 0 0 0 0
0 1 0 0 0 0 0
0 0 1 0 0 0 0
0 0 0 1 0 0 0
0 0 0 0 1 0 0
0 0 0 0 0 1 0

[OUTPUT_PATTERNS]
1 0 0 0 0 0 0
0 1 0 0 0 0 0
0 0 1 0 0 0 0
0 0 0 1 0 0 0
0 0 0 0 1 0 0
0 0 0 0 0 1 0
// End of file
```

Number of layers of the NN. 1 input layer, 1 hidden layer, and 1 output neuron layer.

Neurons by layer. We recommend you to make a previous study in order to determine the "optimum" number of neurons by layer

Activation Function. For this case a binary neuron activation function (0 and 1) is used. Other possible values are sigmoid, linear, etc.

Number of patterns in the training set. i.e., in this file

Input patterns. The numeric values input to the NN. They represent the problem inputs, one by line

Output patterns. Expected output for each on the above input patterns



In a similar way, we can format the test pattern file. For this case however, you should avoid any specification for the sections `ACTIVATION_FUNCTION` and `NEURONS_BY_LAYER`, because this information was recorded by NN C++ class object when loading the train pattern set just described.

File OCHO383_TEST.NET

```
[NUMBER_OF_PATTERNS]
2

[INPUT_PATTERNS]
0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 1

[OUTPUT_PATTERNS]
0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 1

// End of file
```

Notice that in this typical test file the number of required sections are reduced to three, since the rest of information has been defined for the training process.

Therefore, we have just described the architecture of the NN and the data sets used to train and to test the results of the constructed NNs. But, in order to construct (train and test) the best possible NN we need a learning algorithm. There exist many alternative techniques for this task, such as backpropagation of the errors, but heuristic techniques are continuously being proposed with success.

Next, we discuss a brief description of both sequential and distributed models in the following sections.



Sequential Model

The `seq_model` directory contains the source code and configuration files to run the NN training problem using EPOL in sequential. Before running this example, let us have a look to the required files:

- `xxx.param` file. This file contains all the parameters needed to configurate EPOL. This file contains some sections, the most important ones are: `NUMBER_OF_OPERATOR`, `SEED` (initialize the random seed used by the GA in its randomized operations), `OPERATORS`, `PROBLEM_CONFIG_FILE`, `NUMERIC_RANGES`, and `OUTPUT_DISPLAY`. For this example we will use the file `seq_decoder_ga.param`
- `xxx.net` files. As we mention above, for each NN problem to be solved in EPOL we will need 2 `.net` files, namely one for training and one for testing the pattern set. Our training and evaluation pattern sets for the encoded example are, respectively: `OCHO383_train.net` and `OCHO383_test.net`.
- `xxx.out` file. This file is created every time the `OUTPUT_DISPLAY` section from the `.param` file is set to `ALL`. The file contains the results of a trace of execution. It's recommended to set this param when debugging the behaviour of the EA.

Now that we have presented the main files for this example, we need to implement the GA algorithm in a `.cpp` file using the classes and services of EPOL. To reach this goal for the NN training problem just open the file `main_decoder_ga.cpp` file to see how it implements the example at hands (a simple GA that trains a NN for the encoder problem). Follow the instructions made in the comments.

Once you understand completely all the files mentioned before, it's time to compile and generate an executable file. For sequential projects, use the `makefile` in the directory `seq_model`. Write at the prompt of your system: `"makefile -f ga_example"`. You should have installed a `g++` compiler with version greater than 2.0, because we are using templates that might cause some problems in earlier versions.

Now the system is ready. Just execute the file `ga_decoder.exe` and wait until EPOL ends. A report will be issued to your console and a result file will be generated.

Authors' note: EPOL has been designed to be executed under Linux or Unix machines with `g++` compiler. At this moment, we do not know of any incompatibility or bug for it. Anyway, if you detect any problem with EPOL, please, do not hesitate to write us with the error description you got. We will provide a solution to it in our Web page as soon as possible.



Distributed Model

A distributed model is an extension of the basic centralized model. You may want to analyze a distributed model to solve problems with your sequential or centralized algorithm such as premature convergence to local optima, high computational times, or just to investigate new models of optimisation.

The main differences between the sequential and the distributed EA models can be found in the way by which the evolutionary algorithm evolves. In the following, we are going to learn how to configure a distributed island model (the one EPOL uses). It is assumed that the population of the sequential model is distributed in several sub-populations that undergo separate evolution to search a solution for the same problem. These sub-algorithms sparsely exchange information to collaborate in this task in a ring of processes.

Here, we only discuss the differences (the additionally needed files) with respect to the sequential versions. The first additional file we must take care of is the island implementation file. This file is named `island_decoder.cpp`; it contains the definition and implementation of a real coded EA. The island configuration file is `island.par`. For simplicity, we assume that the islands share the same configuration file, but this is just a simplification. End users may want to use different ones to conduct an heterogeneous search.

Yet another important file to mention is the `monitor_decoder.cpp` file. This file contains the definition and implementation of a distributed ring of processes with a monitor process that internally manages the parallel computations. Our monitor reads the `monitor.par` file which contains the necessary information to execute it properly.

Basically, these are the most important files we need. Now, in order to execute the distributed model, we need to compile. For this purpose, type `makefile -f all`. This sentence creates the executable files. Now execute `monitor.exe` and wait for the results.