DAta Mining & Exploration Program

Dipartimento di Scienze Fisiche
Università di Napoli "Federico II"

ISTITUTO NAZIONALE di ASTROFISICA
OSSERVATORIO ASTRONOMICO di CAPODIMONTE

CALTECH

Fast Multi Layer Perceptron with Genetic Algorithm

User Manual

DAME-MAN-NA-0012

Issue: 1.3
Author: M. Brescia, A. Solla

Doc. : FMLPGA_UserManual_DAME-MAN-NA-0012-Rel1.3

INDEX

TABLE INDEX

FIGURE INDEX

1 Introduction

The present document is the user guide of the data mining model FMLPGA (Fast Multi Layer Perceptron trained by Genetic Algorithms), as implemented and integrated into the DAMEWARE. This manual is one of the specific guides (one for each data mining model available in the webapp) having the main scope to help user to understand theoretical aspects of the model, to make decisions about its practical use in problem solving cases and to use it to perform experiments through the webapp, by also being able to select the right functionality associated to the model, based upon the specific problem and related data to be explored, to select the use cases, to configure internal parameters, to launch experiments and to evaluate results.

The documentation package consists also of a general reference manual on the webapp (useful also to understand what we intend for association between functionality and data mining model) and a GUI user guide, providing detailed description on how to use all GUI features and options.

So far, we strongly suggest to read these two manuals and to take a little bit of practical experience with the webapp interface before to explore specific model features, by reading this and the other model guides.

All the cited documentation package is available from the address
http://dame.dsf.unina.it/dameware.html**, where there is also the direct gateway to the webapp.**

As general suggestion, the only effort required to the end user is to have a bit of faith in Artificial Intelligence and a little amount of patience to learn basic principles of its models and strategies.

By merging for fun two famous commercial taglines we say: "*Think different, Just do it!*"
(casually this is an example of *data* (*text) mining*...!)

2 FMLPGA Model Theoretical Overview

This paragraph is intended to furnish a theoretical overview of the FMLPGA model, associated to single or multiple functionality domains, in order to be used to perform practical scientific experiments with such techniques. An overview of machine learning and functionality domains, as intended in DAME Project can be found in [A18].

2.1 Multi Layer Perceptron

The MLP architecture is one of the most typical *feed-forward* neural network model. The term feed-forward is used to identify basic behavior of such neural models, in which the impulse is propagated always in the same direction, e.g. from neuron input layer towards output layer, through one or more hidden layers (the network brain), by combining weighted sum of *weights associated to all neurons* (except the input layer).

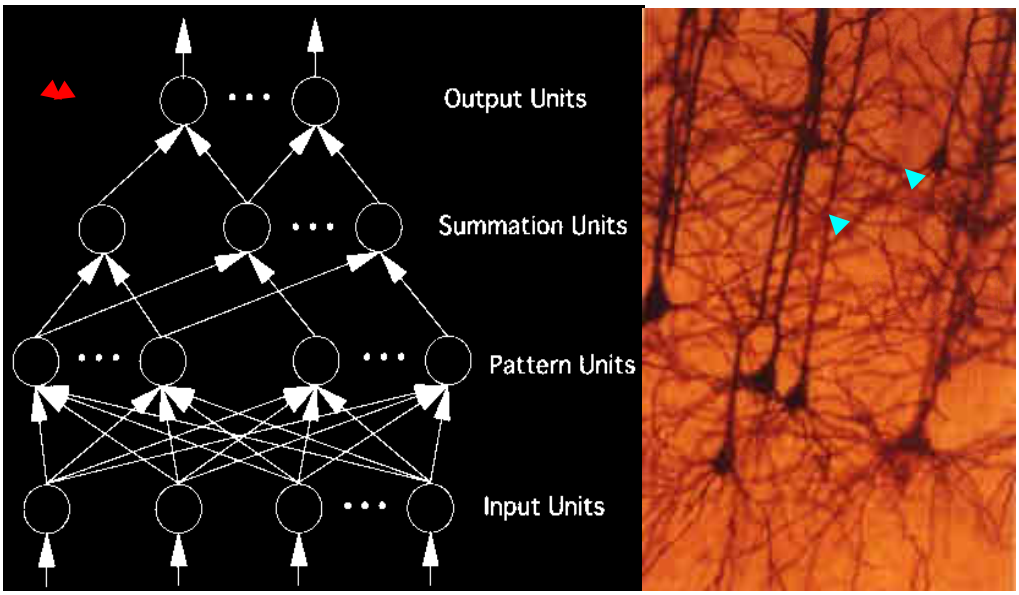


Fig. 1 – the MLP artificial and biologic brains

As easy to understand, the neurons are organized in layers, with proper own role. The input signal, simply propagated throughout the neurons of the input layer, is used to stimulate next hidden and output neuron layers. The output of each neuron is obtained by means of an *activation function*, applied to the weighted sum of its inputs. Different shape of this activation function can be applied, from the simplest *linear* one up to *sigmoid, or tanh* (or a customized function ad hoc for the specific application).

Step function

y = 1 if x >= 0, 0 if x < 0

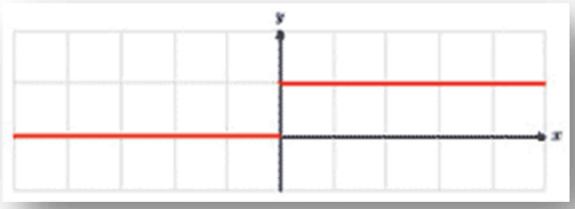


Fig. 2 – the step function

The step function is the most similar to biological neuron reaction to external stimuli. In this case in fact, it uses a constant activation threshold to propagate the signal. It is useful only in problem solving where it is needed a crisp classification between two well identified classes.

Ramp function

y = x if x is internal to [-1, 1]

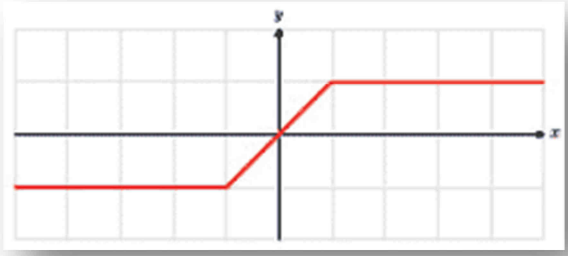


Fig. 3 – the ramp function

It is also known as identity function in the [-1, 1] range. Useful only when the network output is unbounded.

Sigmoid function

y = 1/(1+e^(-x))

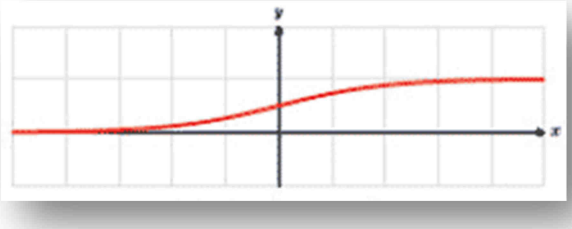


Fig. 4 – the sigmoid function

This function is the most frequent in the MLP model. It is characterized by its smooth step between 0 and 1 with a variable threshold. But this restricted domain [0, 1] is also its limitation. It in fact can be used only in problems where expected outputs are numbers in this range.

Hyperbolic tangent function

y = tanh(x)

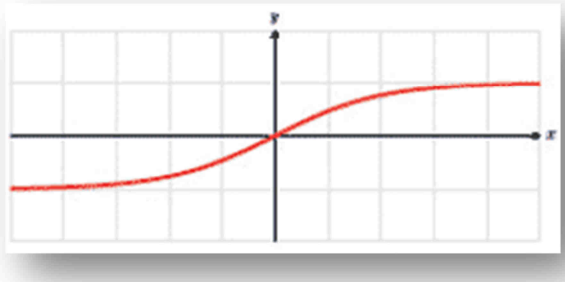


Fig. 5 – the tanh function

The hyperbolic tangent is very similar to the sigmoid except for the output range [-1, 1] and that it across the origin of axes. It therefore extends the admissible range of network output[1].

For the output units, activation functions suited to the distribution of the target values are:

- For binary (0/1) targets, the logistic sigmoid function is an excellent choice;
- For continuous-valued targets with a bounded range, the logistic and hyperbolic tangent functions can be used, where you either scale the outputs to the range of the targets or scale the targets to the range of the output activation function ("scaling" means multiplying by and adding appropriate constants);
- If the target values are positive but have no known upper bound, you can use an exponential output activation function, but you must beware of overflow;
- For continuous-valued targets with no bounds, use the identity or "linear" activation function (which amounts to no activation function) unless you have a very good reason to do otherwise.

The base of the MLP is the **Perceptron,** a type of artificial neural network invented in 1957 at the Cornell Aeronautical Laboratory by Frank Rosenblatt. It can be seen as the simplest kind of feedforward neural network: a linear classifier. The Perceptron is a binary classifier which maps its input *x* (a real-valued vector) to an output value *f*(*x*) (a single binary value) across the matrix.

$$f(x) = \begin{cases} 1 & \text{if } w \cdot x + b > 0 \\ 0 & \text{else} \end{cases}$$

[1] Despite of the above considerations, it is well known that it is always possible to shift the center of the activation function by introducing the bias value associated to each neuron. By this way it is always possible to obtain values in the [–∞, +∞] range. But in this case it is suggested to use a linear transfer function.

where *w* is a vector of real-valued weights and $w \cdot x$ is the dot product (which computes a weighted sum). *b* is the 'bias', a constant term that does not depend on any input value. The value of *f*(*x*) (0 or 1) is used to classify *x* as either a positive or a negative instance, in the case of a binary classification problem. If *b* is negative, then the weighted combination of inputs must produce a positive value greater than | *b* | in order to push the classifier neuron over the 0 threshold. Spatially, the bias alters the position (though not the orientation) of the decision boundary.
The Perceptron learning algorithm does not terminate if the learning set is not linearly separable. The Perceptron is considered the simplest kind of feed-forward neural network.
The earliest kind of neural network is a S*ingle Layer Perceptron* (SLP) network, which consists of a single layer of output nodes; the inputs are fed directly to the outputs via a series of weights. In this way it can be considered the simplest kind of feed-forward network. The sum of the products of the weights and the inputs is calculated in each node, and if the value is above some threshold (typically 0) the neuron fires and takes the activated value (typically 1); otherwise it takes the deactivated value (typically -1).

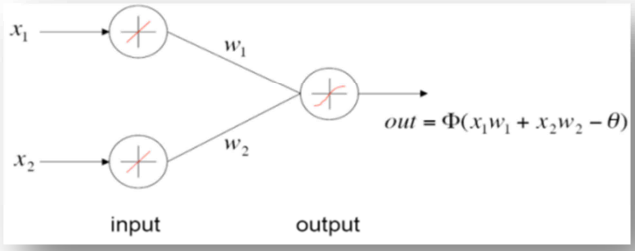


Fig. 6 – Example of a SLP to calculate the logic AND operation

Neurons with this kind of activation function are also called a*rtificial neurons* or *linear threshold units*, as described by Warren McCulloch and Walter Pitts in the 1940s.
A Perceptron can be created using any values for the activated and deactivated states as long as the threshold value lies between the two. Most perceptrons have outputs of 1 or -1 with a threshold of 0 and there is some evidence that such networks can be trained more quickly than networks created from nodes with different activation and deactivation values. SLPs are only capable of learning linearly separable patterns. In 1969 in a famous monograph entitled *Perceptrons* Marvin Minsky and Seymour Papert showed that it was impossible for a single-layer Perceptron network to learn an XOR function.
Although a single threshold unit is quite limited in its computational power, it has been shown that networks of parallel threshold units can approximate any continuous function from a compact interval of the real numbers into the interval [-1,1]. So far, it was introduced the model Multi Layer Perceptron.

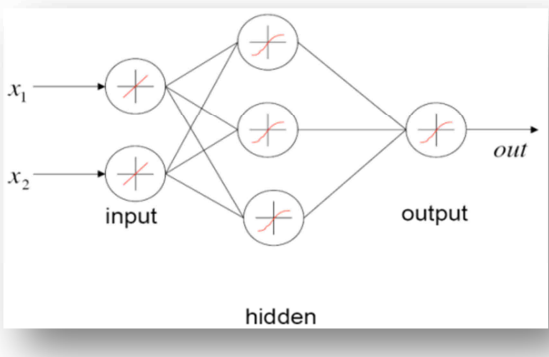


Fig. 7 – A MLP able to calculate the logic XOR operation

This class of networks consists of multiple layers of computational units, usually interconnected in a feed-forward way. Each neuron in one layer has directed connections to the neurons of the subsequent layer. In many applications the units of these networks apply a continuous activation function.

The number of hidden layers represents the degree of the complexity achieved for the energy solution space in which the network output moves looking for the best solution. As an example, in a typical classification problem, the number of hidden layers indicates the number of hyper-planes used to split the parameter space (i.e. number of possible classes) in order to classify each input pattern.
The *universal approximation theorem* [R12] for neural networks states that every continuous function that maps intervals of real numbers to some output interval of real numbers can be approximated arbitrarily closely by a multi-layer Perceptron with just one hidden layer. This result holds only for restricted classes of activation functions, e.g. for the sigmoidal functions.
An extension of the universal approximation theorem states that the two layers architecture is capable of universal approximation and a considerable number of papers have appeared in the literature discussing this property. An important corollary of these results is that, in the context of a classification problem, networks with sigmoidal non-linearity and two layer of weights can approximate any decision boundary to arbitrary accuracy. Thus, such networks also provide universal non-linear discriminate functions. More generally, the capability of such networks to approximate general smooth functions allows them to model posterior probabilities of class membership. Since two layers of weights suffice to implement any arbitrary function, one would need special problem conditions, or requirements to recommend the use of more than two layers. Furthermore, it is found empirically that networks with multiple hidden layers are more prone to getting caught in undesirable local minima.
Astronomical data do not seem to require such level of complexity and therefore it is enough to use just a double weights layer, i.e. a single hidden layer.
What is different in such a neural network architecture is typically the learning algorithm used to train the network. It exists a dichotomy between *supervised* and *unsupervised* learning methods.
In the first case, the network must be firstly trained (*training phase*), in which the input patterns are submitted to the network as couples (input, desired known output). The feed-forward algorithm is then achieved and at the end of the input submission, the network output is compared with the corresponding desired output in order to quantify the learning quote. It is possible to perform the comparison in a *batch* way (after an entire input pattern set submission) or *incremental* (the comparison is done after each input pattern submission) and also the *metric* used for the *distance* measure between desired and obtained outputs, can be chosen accordingly problem specific requirements (usually the euclidean distance is used).
After each comparison and until a desired error distance is unreached (typically the error tolerance is a pre-calculated value or a constant imposed by the user), the weights of hidden layers must be changed accordingly to a particular law or learning technique.
After the training phase is finished (or arbitrarily stopped), the network should be able not only to recognize correct output for each input already used as training set, but also to achieve a certain degree of *generalization*, i.e. to give correct output for those inputs never used before to train it. The degree of generalization varies, as obvious, depending on how “good” has been the learning phase. This important feature is realized because the network doesn’t associates a single input to the output, but it discovers the relationship present behind their association. After training, such a neural network can be seen as a black box able to perform a particular function (input-output correlation) whose analytical shape is a priori not known. In order to gain the best training, it must be as much homogeneous as possible and able to describe a great variety of samples. Bigger the training set, higher will be the network generalization capability.
Despite of these considerations, it should be always taken into account that neural networks application field should be usually referred to problems where it is needed high flexibility (quantitative result) more than high precision (qualitative results).
Second learning type (unsupervised) is basically referred to neural models able to classify/cluster patterns onto several categories, based on their common features, by submitting training inputs without related

desired outputs. This is not the learning case approached with the MLP architecture, so it is not important to add more information in this document.

2.2 The Genetic Algorithms

The *Genetic Algorithms* (GAs) are computational methods inspired by the natural evolution law discovered by Darwin. They are particularly powerful to solve problems where the solution space is not well defined. The algorithm hence consists mainly in the cyclic exploration of the parameter space, carefully going towards the best solution. In a GA each element of a population is the so-called *chromosome*, composed by a set of *genes* (features), that represents its *DNA*. Each DNA is in practice one possible solution to the problem. The starting point of the method consists in the random generation of a population of chromosomes, for example by using normal or uniform statistical distributions. The method proceeds by performing cyclic variation and combination of the initial population, looking for the best population (best problem solution). At each evolution stage the output chromosomes are obtained by applying several *genetic operators* to the input population and by evaluating through a specific *fitness* function the goodness of the new generated population. The fitness function has the basic role to give a method to discard worst chromosomes from the population, achieving the evolution to the next generation of the best candidates only (exactly like Nature works with its species, following the Darwin's law). Typical genetic operators are cross-over and mutation.

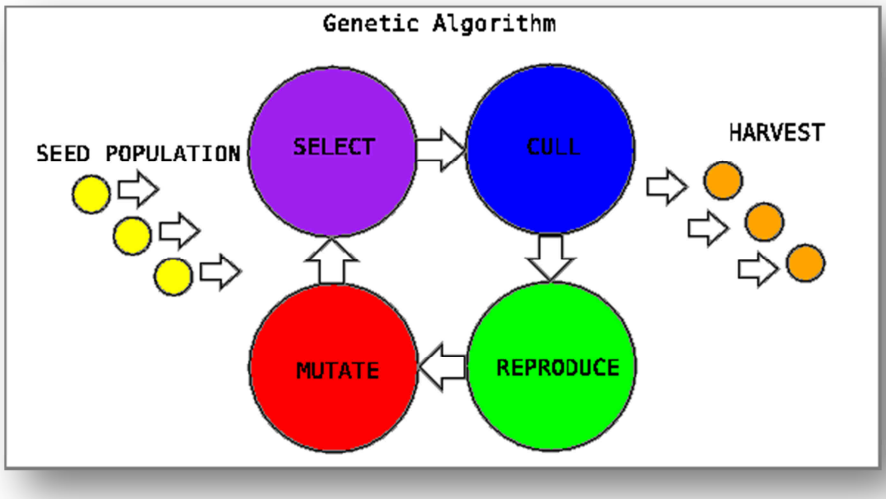


Fig. 8 – A typical Genetic Algorithm optimization research mechanism

In the design of a GA to solve an optimization problem, three steps are considered as strategic to obtain better results in the better time: the chromosome representation (codified in some way); the choice of the fitness function and the method to choose chromosome reproduction. The latter in particular deals with the issue that no random choices can be applied to select chromosomes to be combined by the genetic operator employed in the algorithm. Otherwise, the convergence could result too slow. The choice must be "driven" in some way. Usual rules are the so-called *roulette wheel selection* and *tournament selection*. The former mainly consists into the assignment of a probability to be selected for the reproduction for each chromosome. This probability is directly proportional to its fitness. The latter is alternatively based on the random choice of a number N of chromosomes from the current population and to compare their fitness: the element with the best fitness is the winner and is selected for reproduction.

The tournament selection seems quite flexible, because it permits to introduce more variability in the selection criterion: higher N, lower the possibility to select chromosomes candidates with worst fitness!

Anyway, not always the elements with worst fitness must be considered bad choices for reproduction. They in fact, introduce more variety in the population (exactly like the jumping out from a local minima for the gradient descendent learning algorithm in the BP…), and can speed up the convergence towards best solution. But their use must be taken under control in the GA. For example by monitoring population evolution, where some chromosome has a static trend (i.e. it has always associated a good fitness but not sufficient to become a winner during reproduction, so remaining always the same through several generations). In this case, by applying any reproduction with a chromosome randomly chosen between those with poor fitness, can introduce gain in the next generations.

Usually (but not always) in the tournament selection two winners are chosen at a time (i.e. from two tournaments) and recombined to generate two sons put in the next generation. As said before, two are typical genetic operators employed in the chromosome recombination:

The ***Cross-over*** (or unbiased chromosome cross-over) criterion, on two m-length chromosomes A and B, brakes original parents at a fixed position j (j is the number of genes or sub-string length arbitrarily fixed a priori). So the m-j genes of chromosome A are queued to the first j genes of chromosome B, while the m-j genes of chromosome B are queued to the first j genes of chromosome A.

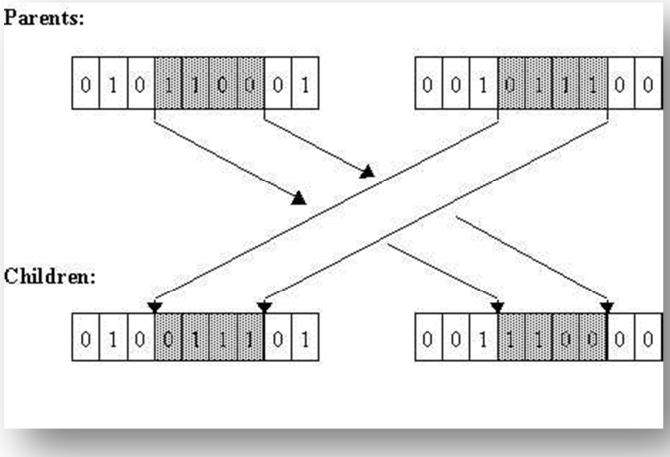


Fig. 9 – An example of genetic cross over operator application

Example:

A = 10010001, B = 00110111, m = 8 and j = 5 => after cross-over => A' = 10010111, B' = 00110001

After the cross-over application, the two sons A' and B' are obtained from their parents by reciprocally reverting last m-j (3) genes.

After cross-over, usually it is applied another genetic operator, ***mutation***. It consists into the roughly change in only one gene inside the chromosome. Depending on the code used to represent the chromosome genes (typical is the binary code), the original value of a gene is replaced by the other character. There are two different mutation criteria: unbiased and biased. Both criteria can be applied to an entire chromosome as well as to a single gene.

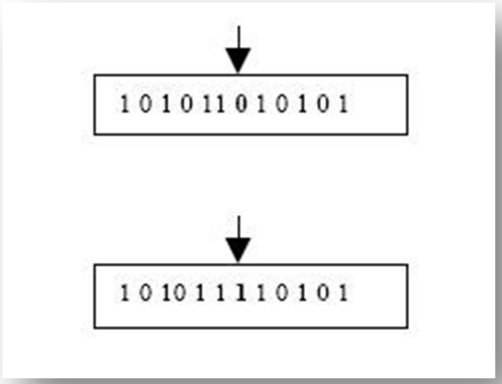


Fig. 10 – An example of genetic mutation operator application

In the *single gene unbiased mutation* (standard mutation type), one position inside the chromosome string is randomly selected. Then the gene related to the selected chromosome string position is replaced. In the *unbiased chromosome mutation*, the entire selected chromosome is completely replaced by a new chromosome, randomly generated. In the *biased chromosome mutation*, the selected chromosome is totally replaced by a new chromosome, obtained by the sum of original chromosome and a new one randomly generated (in this sense biased).

Generally, neither the cross-over and mutation operators are always applied to all population members, but the first usually with a certain pre-defined probability (typical is 70% of population members), while the second with a lower probability and only to members with associated poor fitness.

This procedure (serial application of selected genetic operators) is applied until the new population has exactly the same member number of the previous one. And the entire cycle of population generation is iterated until the chromosome with desired good fitness is founded.

Now, having introduced the computing model architecture and the learning algorithm, we are ready to combine them into the FMLPGA model.

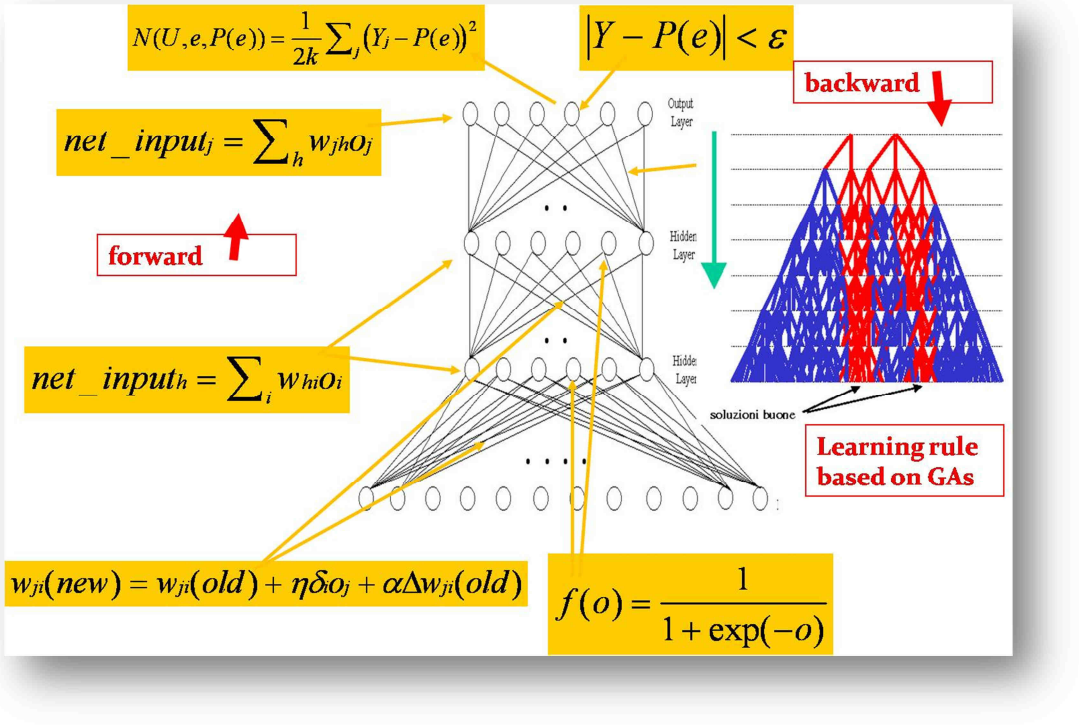


Fig. 11 – A MLP network trained by a Genetic Algorithm

In the DM model the GA technique is used to change weights related to hidden layers during the learning phase (corresponding to the backward step of BP algorithm). In this case the chromosomes are represented by the weight vectors associated with neuron layers, where the single neuron weight is a gene of the string. By evolving populations of network weights, through the application of described genetic operators, the model is able to converge (generally faster than weight gradient descendent of BP) to the best solution. In other words, at each step of genetic evolution, a population of neural networks (identified by its weight vectors) is generated. The fitness function in this case is simply the calculated error of MLP output by applying standard feed-forward propagation of the input patterns through the network layers. The winner chromosome will be therefore the MLP weight configuration obtaining the lower output error. Moreover, in this case, to maintain the integrity of the MLP feed-forward calculations, the genes are not coded but leaved exactly as weight double values (usually normalized between -1 and 1).

The formulas showed in Fig. 11 are cyclically repeated during training. It is hence evident that the learning algorithm can be divided into two phases: bottom-up propagation and top-down weight update.

The **complete algorithm** involves the following steps:

1. Random generation of the initial genetic population of chromosomes (population of different network weight vectors);
2. Forward propagation of a training pattern's input through the neural network in order to generate the propagation's output activations. This step is repeated for each network weight matrix (a single chromosome of the genetic population);
3. Calculation of training error (batch mode) as MSE of distances between network output and related target values;
4. Evaluation of the output performance on all patterns against the genetic fitness function (for each chromosome);

5. Generation of a new population of chromosomes, by applying genetic operators to the weight vectors of the network;
6. Repeat steps from 2 to 5 until the number of training cycles or the chosen error threshold has been reached.

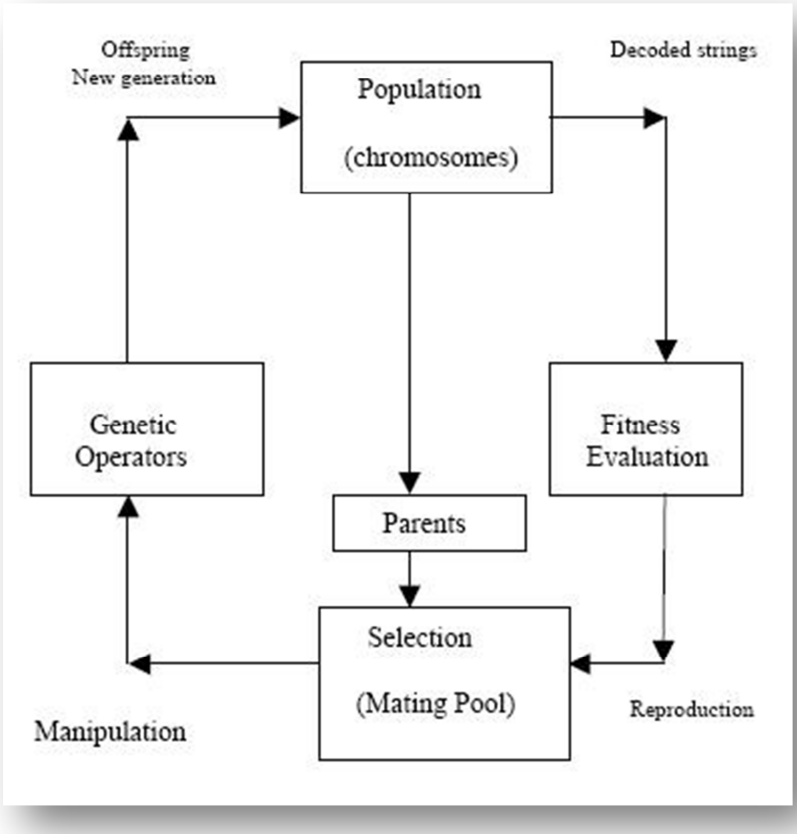


Fig. 12 – Steps of the algorithm related to genetic algorithm evolution

2.3 MLP Practical Rules

The practice and expertise in the machine learning models, such as MLP, are important factors, coming from a long training and experience within their use in scientific experiments. The speed and effectiveness of the results strongly depend on these factors. Unfortunately there are no magic ways to a priori indicate the best configuration of internal parameters, involving network topology and learning algorithm.
But in some cases a set of practical rules to define best choices can be taken into account.

2.3.1 Selection of neuron activation function

- If there are good reasons to select a particular activation function, then do it
  - linear;
  - threshold;
  - Hyperbolic tangent;

  - sigmoid;
- General “good” properties of activation function
  - Non-linear;
  - Saturate – some max and min value;
  - Continuity and smooth;
  - Monotonicity: convenient but nonessential;
  - Linearity for a small value of net;
- Sigmoid function has all the good properties:
  - Centered at zero;
  - Anti-symmetric;
  - f(-net) = - f(net);
  - Faster learning;
  - Overall range and slope are not important;

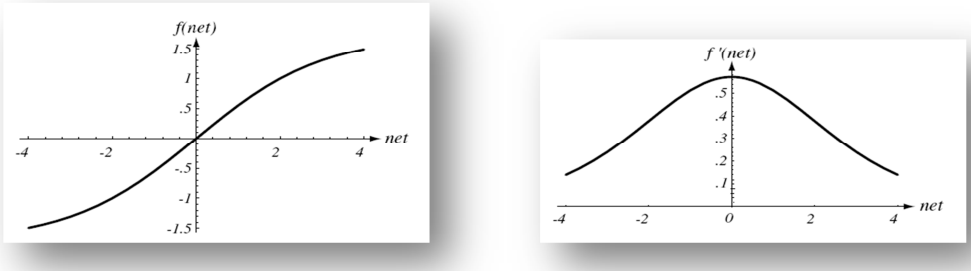


Fig. 13 – The sigmoid function and its first derivative

2.3.2 Scaling input and target values

- Standardize
  - Large scale difference
    - error depends mostly on large scale feature;
  - Shifted to Zero mean, unit variance
    - Need to be done once, before training;
    - Need full data set;
- Target value
  - Output is saturated
    - In the training, the output never reach saturated value;
      - Full training never terminated;
  - Range [-1, +1] is suggested;

2.3.3 Number of hidden nodes

- Number of hidden units governs the expressive power of net and the complexity of decision boundar y;
- Well-separated → fewer hidden nodes;
- From complicated density, highly interspersed → many hidden nodes;
- Heuristics rule of thumb:
  - Use a minimum of 2N+1 neurons of the first hidden layer (N is the number of input nodes);
  - More training data yields better result;

- Number of weights < number of training data;
- Number of weights ≈ (number of training data)/10;
- Adjust number of weights in response to the training data:
  - Start with a "large" number of hidden nodes, then decay, prune weights…;

2.3.4 Number of hidden layers

- One or two hidden layers are OK, so long as differentiable activation function;
  - But one layer is generally sufficient;
- More layers → more chance of local minima;
- Single hidden layer vs double (multiple) hidden layer:
  - single is good for any approximation of continuous function;
  - double may be good some times;
- Problem-specific reason of more layers:
  - Each layer learns different aspects;

3 Use of the web application model

The Multi Layer Perceptron (MLP) is one of the most common supervised neural architectures used in many application fields. It is especially related to classification and regression problems, and in DAME it is designed to be associated with such two functionality domains. The description of these two functionalities is reported in the Reference Manual [A18], available from webapp menu or from the beta intro web page.

In the following are described practical information to configure the network architecture and the learning algorithm in order to launch and execute science cases and experiments.
The GA learning rule is one of the options as available in DAME webapp to be associated with the MLP network.

3.1 The fastest parallel GPU-based version (FMLPGA)

The last version released on DAMEWARE exploits the new HW infrastructure, based on HPC in the Cloud with NVIDIA GPU K20 Kepler series parallel execution capabilities.
The new FMLPGA algorithm (now called FMLPGA, where F stands for Fast) offers the possibility to be executed on two optional (user selected) computing platforms, in a transparent way to end user:

- CPU: normal serialized execution on a HPC host, based on a multi-processor multi-core machinery;
- GPU: accelerated parallel execution on a GPU device, based on a NVIDIA K20 many-core machinery;

The GPU option gives an average speedup acceleration of about 8x in respect of the CPU one. It means that on average a same experiment is executed 8 times faster.

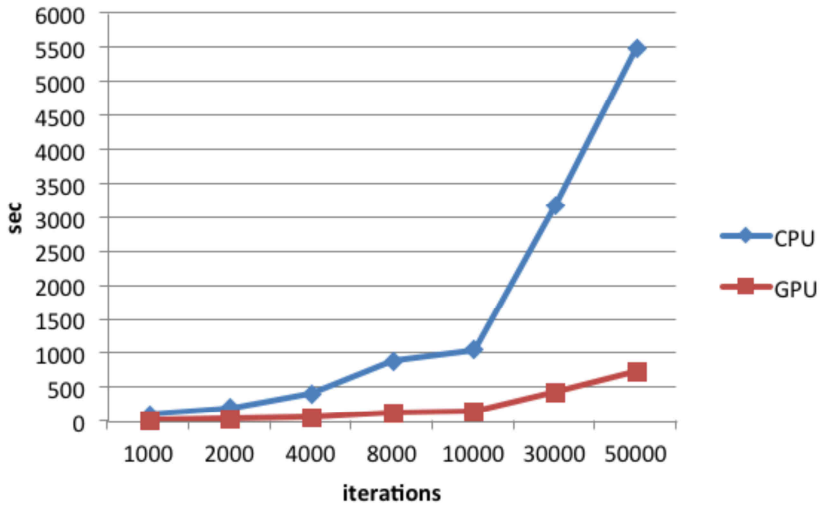


Fig. 14 – The computing time comparison between FMLPGA training execution on CPU and GPU platforms. The experiment was based on 1000 input patterns, each one composed by 10 parameters.

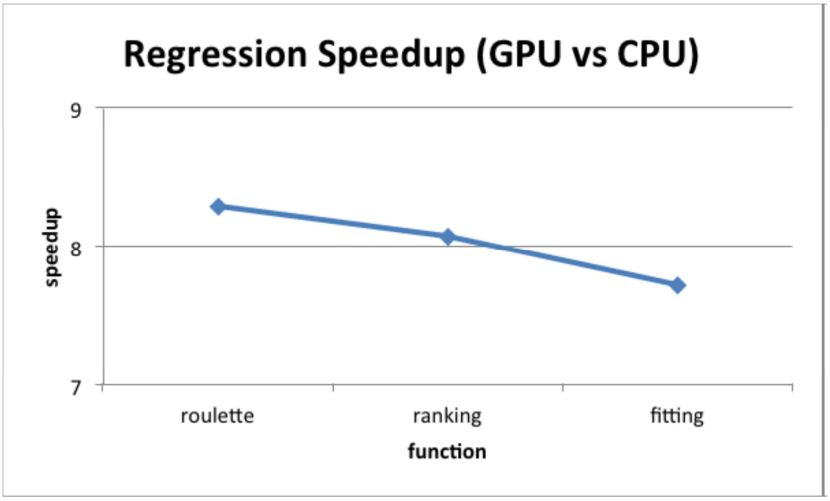


Fig. 15 – The computing time speedup for FMLPGA on CPU and GPU platforms. The function axis is referred to the different chromosome selection type used during training (evolution of genetic population)

The user has the opportunity to select the platform through a simple parameter during experiment setup. For more instructions on parameter setup see below.

3.2 Use Cases

For the user the FMLPGA system offers four use cases:

- ***Train***
- ***Test***
- ***Run***

As described in [A19] a supervised machine learning model like FMLPGA requires different use cases, well ordered in terms of execution sequence. A typical complete experiment with this kind of models consists in the following steps:

1. **Train** the network with a dataset as input, containing both input and target features; then store as output the final weight matrix (best configuration of network weights);
2. **Test** the trained network, in order to verify training quality (it is also included the validation step, available for some models). The same training dataset or a mix with new patterns can be used as input;
3. **Run** the trained and tested network with datasets containing ONLY input features (without target ones). In this case new or different input data are encouraged, because the Run use case implies to simply execute the model, like a generic static function.

3.3 Input

We also remark that massive datasets to be used in the various use cases are (and sometimes must be) different in terms of internal file content representation. Remind that in all DAME models it is possible to use one of the following data types:

- ASCII (extension .dat or .txt): simple text file containing rows (patterns) and columns (features) separated by spaces, normally without header;
- CSV (extension .csv): Comma Separated Values files, where columns are separated by commas;
- FITS (extension .fits): tabular fits files;
- VOTABLE (extension .votable): formatted files containing special fields separated by keywords coming from XML language, with more special keywords defined by VO data standards;

For training and test cases a correct dataset file must contain both input and target features (columns), with input type as the first group and target type as the final group.

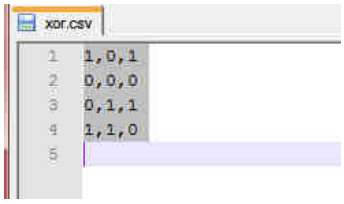


Fig. 16 – The content of the xor.csv file used as input for training/test use cases

As shown in Fig. 14, the xor.csv file for training/test uses cases has 4 patterns (rows) of 2 input features (first two columns) and one target feature (third column). The target feature is not an input information but the desired output to be used in the comparison (calculation of the error) with the model output during a training/test experiment.

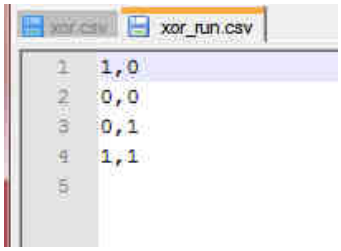


Fig. 17 – The content of the xor_run.csv file used as input for Run use case

In Fig. 15, the xor_run.csv file is shown, valid only for Run use case experiments. It is the same of xor.csv except for the target column that is not present. This file can be also generated by the user starting from the xor.csv. As detailed in the GUI user Guide [A19], the user may in fact use the Dataset Editor options of the webapp to manipulate and build datasets starting from uploaded data files.
IMPORTANT NOTE: in case of classification experiment, the target columns of input data file must be at least two (2-class problem) or more than two!

3.4 Output

In terms of output, different files are obtained, depending on the specific use case of the experiment.
In the case of **regression** functionality, the following output files are obtained in all use cases:

TRAIN default prefix: FFMLPGA_Train	**TEST default prefix: FFMLPGA_Test**	**RUN default prefix: FFMLPGA_Run**	**NOTES**
_params.xml	_params.xml	_params.xml	experiment internal configuration info
.log	.log	.log	experiment status log
_trainerrors.txt			error trend table
_trainerrors.jpeg			error trend plot
_trainoutput.csv	_testoutput.csv	_runoutput.csv	output
_weights.txt			trained network weights (to be moved in the File Manager tab area through GUI button AddInWS, to be loaded during a test/run experiment)
_trainconfusionmatrix.txt			pseudo confusion matrix (statistical evaluation of training performance)
	_testconfusionmatrix.txt		pseudo confusion matrix (statistical evaluation of test performance)

Tab. 1 – output file list in case of regression type experiments

In the case of **classification** functionality, the following output files are obtained in all use cases:

TRAIN default prefix: FFMLPGA_Train	**TEST default prefix: FFMLPGA_Test**	**RUN default prefix: FFMLPGA_Run**	**NOTES**
_params.xml	_params.xml	_params.xml	experiment internal configuration info
.log	.log	.log	experiment status log
_trainerrors.txt			error trend table
_trainerrors.jpeg			error trend plot
_trainoutput.csv	_testoutput.csv	_runoutput.csv	output
_weights.txt			trained network weights (to be moved in the File Manager tab area through GUI button AddInWS, to be loaded during a test/run experiment)
_trainconfusionmatrix.txt			confusion matrix (statistical evaluation of training performance)
	_testconfusionMatrix.txt		confusion matrix (statistical evaluation of test performance)

Tab. 2 – output file list in case of classification type experiments

3.5 TRAIN Use case

In the use case named "**Train**", the software provides the possibility to train the FMLPGA. The user will be able to use new or existing (already trained) MLP weight configurations, adjust parameters, set training parameters, set training dataset, manipulate the training dataset and execute the training experiments.

There are several parameters to be set to achieve training, dealing with network topology and learning algorithm. In the experiment configuration there is also the Help button, redirecting to a web page dedicated to support the user with deep information about all parameters and their default values.

We remark that all parameters labeled by an asterisk are considered required. In all other cases the fields can be left empty (default values are used).

3.5.1 Regression with FMLPGA – Train Parameter Specifications

In the case of Regression_FMLPGA with Train use case, the help page is at the address:
http://dame.dsf.unina.it/FMLPGA_help.html#regr_train

- **input dataset**

this parameter is a field required!
This is the dataset file to be used as input for the learning phase of the model. It typically must include both input and target columns, where each row is an entire pattern (or sample of data). The format (hence its extension) must be one of the types allowed by the application (ASCII, FITS, CSV, VOTABLE). More specifically, take in mind the following simple rule: the sum of input and output nodes MUST be equal to the total number of the columns in this file!

- **GPU or CPU**

It is a file generated by the model during training phase. It contains the resulting network topology as stored at the end of a training session. Usually this file should not be edited or modified by users, just to preserve its content as generated by the model itself.
Accepted entries are:

0: serial type execution (on CPU)
1: parallel type execution (on GPU)

If left empty, its default is 1 (GPU)

- **input nodes**

this parameter is a field required!
It is the number of neurons at the first (input) layer of the network. It must exactly correspond to the number of input columns in the dataset input file (Training File field), except the target columns.

- **hidden layers**

It is the number of hidden layers of the MLP network. It can assume only two values (1 or 2). As suggestion this should be selected according the complexity of the problem (usually 1 layer would be sufficient).

If left empty, its default is 1

- **1st hidden layer nodes**

this parameter is a field required!
It is the number of neurons of the first hidden layer of the network. As suggestion this should be selected in a range between a minimum of 2N + 1, where N is the number of input nodes.

- **1st activation function**

It is the choice of which activation function should be associated to neurons of the 1st hidden layer. After this choice, all neurons of the layer will use the same activation function type. The options are:

0: sigmoid
1: threshold
2: linear
3: hyperbolic tangent

By default, the hyperbolic tangent function is used.

- **2nd hidden layer nodes**

It is the number of neurons of the second hidden layer of the network. As suggestion this should be selected smaller than the previous layer.
By default the second hidden layer is empty (not used)

- **2nd activation function**

It is the choice of which activation function should be associated to neurons of the 2nd hidden layer. After this choice, all neurons of the layer will use the same activation function type. The options are:

0: sigmoid
1: threshold
2: linear
3: hyperbolic tangent

By default, if the 2nd layer is activated, the hyperbolic tangent function is used.

- **output activation function**

It is the choice of which activation function should be associated to neurons of the output layer. After this choice, all neurons of the layer will use the same activation function type. The options are:

0: sigmoid
1: threshold
2: linear
3: hyperbolic tangent

By default, the hyperbolic tangent function is used.

- **selection function**

It is the choice of the selection function used to evolve chromosomes of genetic population at each training iteration. The options are:

0: fitting
1: ranking
2: standard roulette (v1)
3: optimized roulette (v2)

By default, the standard roulette function is used.

- **error threshold**

This is the threshold of the learning loop. This is one of the two stopping criteria of the algorithm. Use this parameter in combination with the number of iterations.
If left empty, its default is 0.01

- **epochs**

Number of training epochs, done in batch learning mode. This is the second stop condition of the algorithm.
If left empty, its default is 1000

- **error log frequency**

Frequency (in steps) of training error storage in a log file, to evaluate the trend in the learning error during generation cycles.
If left empty, its default is 10

- **cross over rate**

probability percentage (a real value between 0 and 1) of the cross over operator application during the population generation process.
Crossover happens when two chromosomes "break" themselves at the same point (inside the string coding the gene vector) and "exchange" their segments.
As all genetic operators, the crossover is not always applied in the genetic recombination, but with an associated probability (this parameter). Instead, the breaking point inside the chromosome where to apply crossover is selected randomly
If left empty, its default is 0.9 (i.e. 90%)

- **mutation rate**

probability percentage (a real value between 0 and 1) of the mutation operator application during the population generation process.
The mutation operator makes a single change in a gene of a chromosome, replacing it with a new random value, selected in a fixed range (see parameter "chromosome perturbation value").
If left empty, its default is 0.4 (i.e. 40%)

- **population size**

This is the number of chromosomes which compose the population (an integer between 10 and 60). Remember that each chromosome is a solution of the problem. At each iteration (generation) this parameter indicates how many chromosomes should be considered in the population of the GA.
If left empty, the default value is 20

- **elitism rate**

The parameter (user defined) related to this elitism mechanism defines the number of copies of the winner chromosome to be transmitted unchanged in the population of the next generation.
If left empty, its default is 2

- **tournament participants**

This is the number of chromosomes in the population to be engaged in the so called "Ranking Selection".
This is in practice used only in case of "ranking" selection function choice. Among this number of participants, the first two chromosomes with higher fitness value are chosen to generate childs.

3.5.2 Classification with FMLPGA – Train Parameter Specifications

In the case of Classification_FMLPGA with Train use case, the help page is at the address:
http://dame.dsf.unina.it/FMLPGA_help.html#class_train

- **input dataset**

this parameter is a field required!
This is the dataset file to be used as input for the learning phase of the model. It typically must include both input and target columns, where each row is an entire pattern (or sample of data). The format (hence its extension) must be one of the types allowed by the application (ASCII, FITS, CSV, VOTABLE). More specifically, take in mind the following simple rule: the sum of input and output nodes MUST be equal to the total number of the columns in this file!

- **GPU or CPU**

It is a file generated by the model during training phase. It contains the resulting network topology as stored at the end of a training session. Usually this file should not be edited or modified by users, just to preserve its content as generated by the model itself.
Accepted entries are:

0: serial type execution (on CPU)
1: parallel type execution (on GPU)

If left empty, its default is 1 (GPU)

- **input nodes**

this parameter is a field required!
It is the number of neurons at the first (input) layer of the network. It must exactly correspond to the number of input columns in the dataset input file (Training File field), except the target columns.

- **hidden layers**

It is the number of hidden layers of the MLP network. It can assume only two values (1 or 2). As suggestion this should be selected according the complexity of the problem (usually 1 layer would be sufficient).
If left empty, its default is 1

- **1st hidden layer nodes**

this parameter is a field required!
It is the number of neurons of the first hidden layer of the network. As suggestion this should be selected in a range between a minimum of 2N + 1, where N is the number of input nodes.

- **1st activation function**

It is the choice of which activation function should be associated to neurons of the 1st hidden layer. After this choice, all neurons of the layer will use the same activation function type. The options are:

0: sigmoid
1: threshold
2: linear
3: hyperbolic tangent

By default, the hyperbolic tangent function is used.

- 2nd hidden layer nodes

It is the number of neurons of the second hidden layer of the network. As suggestion this should be selected smaller than the previous layer.
By default the second hidden layer is empty (not used)

- **2nd activation function**

It is the choice of which activation function should be associated to neurons of the 2nd hidden layer. After this choice, all neurons of the layer will use the same activation function type. The options are:

0: sigmoid
1: threshold
2: linear
3: hyperbolic tangent

By default, if the 2nd layer is activated, the hyperbolic tangent function is used.

- **output nodes**

this parameter is a field required!
It is the number of neurons in the output layer of the network. It must correspond to the number of target columns as contained in the dataset input file (filed Training File). **It is mandatory to set 2 neurons at least**.

- **output activation function**

It is the choice of which activation function should be associated to neurons of the output layer. After this choice, all neurons of the layer will use the same activation function type. The options are:

0: sigmoid
1: threshold
2: linear
3: hyperbolic tangent

By default, the hyperbolic tangent function is used.

- **selection function**

It is the choice of the selection function used to evolve chromosomes of genetic population at each training iteration. The options are:

0: fitting
1: ranking
2: standard roulette (v1)
3: optimized roulette (v2)

By default, the standard roulette function is used.

- **error threshold**

This is the threshold of the learning loop. This is one of the two stopping criteria of the algorithm. Use this parameter in combination with the number of iterations.
If left empty, its default is 0.01

- **epochs**

Number of training epochs, done in batch learning mode. This is the second stop condition of the algorithm.
If left empty, its default is 1000

- **error log frequency**

Frequency (in steps) of training error storage in a log file, to evaluate the trend in the learning error during generation cycles.
If left empty, its default is 10

- **cross over rate**

probability percentage (a real value between 0 and 1) of the cross over operator application during the population generation process.
Crossover happens when two chromosomes "break" themselves at the same point (inside the string coding the gene vector) and "exchange" their segments.
As all genetic operators, the crossover is not always applied in the genetic recombination, but with an associated probability (this parameter). Instead, the breaking point inside the chromosome where to apply crossover is selected randomly
If left empty, its default is 0.9 (i.e. 90%)

- **mutation rate**

probability percentage (a real value between 0 and 1) of the mutation operator application during the population generation process.
The mutation operator makes a single change in a gene of a chromosome, replacing it with a new random value, selected in a fixed range (see parameter "chromosome perturbation value").
If left empty, its default is 0.4 (i.e. 40%)

- **population size**

This is the number of chromosomes which compose the population (an integer between 10 and 60). Remember that each chromosome is a solution of the problem. At each iteration (generation) this parameter indicates how many chromosomes should be considered in the population of the GA.
If left empty, the default value is 20

- **elitism rate**

The parameter (user defined) related to this elitism mechanism defines the number of copies of the winner chromosome to be transmitted unchanged in the population of the next generation.
If left empty, its default is 2

- **tournament participants**

This is the number of chromosomes in the population to be engaged in the so called "Ranking Selection".
This is in practice used only in case of "ranking" selection function choice. Among this number of participants, the first two chromosomes with higher fitness value are chosen to generate childs.

3.6 TEST Use case

In the use case named "**Test**", the software provides the possibility to test the FMLPGA. The user will be able to use already trained FMLPGA models, their weight configurations to execute the testing experiments.

In the experiment configuration there is also the Help button, redirecting to a web page dedicated to support the user with deep information about all parameters and their default values.

We remark that all parameters labeled by an asterisk are considered required. In all other cases the fields can be left empty (default values are used).

3.6.1 Regression with FMLPGA – Test Parameter Specifications

In the case of Regression_FMLPGA with Test use case, the help page is at the address:
http://dame.dsf.unina.it/FMLPGA_help.html#regr_test

- **input dataset**

this parameter is a field required!
Dataset file as input. It is a file containing input and target columns.
It must have the same number of input and target columns as for the training input file.

For example, it could be the same dataset file used as the training input file.

- **weights file**

this parameter is a field required!
It is a file generated by the model during training phase. It contains the resulting network topology as stored at the end of a training session. Usually this file should not be edited or modified by users, just to preserve its content as generated by the model itself.

- **GPU or CPU**

It is a file generated by the model during training phase. It contains the resulting network topology as stored at the end of a training session. Usually this file should not be edited or modified by users, just to preserve its content as generated by the model itself.
Accepted entries are:

0: serial type execution (on CPU)
1: parallel type execution (on GPU)

If left empty, its default is 1 (GPU)

- **input nodes**

this parameter is a field required!
It is the number of neurons at the first (input) layer of the network. It must exactly correspond to the number of input columns in the dataset input file (Training File field), except the target columns.

- **hidden layers**

It is the number of hidden layers of the MLP network. It can assume only two values (1 or 2). As suggestion this should be selected according the complexity of the problem (usually 1 layer would be sufficient).
If left empty, its default is 1

- **1st hidden layer nodes**

this parameter is a field required!
It is the number of neurons of the first hidden layer of the network. As suggestion this should be selected in a range between a minimum of 2N + 1, where N is the number of input nodes.

- **1st activation function**

It is the choice of which activation function should be associated to neurons of the 1st hidden layer. After this choice, all neurons of the layer will use the same activation function type. The options are:

0: sigmoid
1: threshold
2: linear
3: hyperbolic tangent

By default, the hyperbolic tangent function is used.

- **2nd hidden layer nodes**

It is the number of neurons of the second hidden layer of the network. As suggestion this should be selected smaller than the previous layer.
By default the second hidden layer is empty (not used)

- **2nd activation function**

It is the choice of which activation function should be associated to neurons of the 2nd hidden layer. After this choice, all neurons of the layer will use the same activation function type. The options are:

0: sigmoid
1: threshold
2: linear
3: hyperbolic tangent

By default, if the 2nd layer is activated, the hyperbolic tangent function is used.

- **output activation function**

It is the choice of which activation function should be associated to neurons of the output layer. After this choice, all neurons of the layer will use the same activation function type. The options are:

0: sigmoid
1: threshold
2: linear
3: hyperbolic tangent

By default, the hyperbolic tangent function is used.

3.6.2 Classification with FMLPGA – Test Parameter Specifications

In the case of Classification_FMLPGA with Test use case, the help page is at the address:
http://dame.dsf.unina.it/FMLPGA_help.html#class_test

- **input dataset**

this parameter is a field required!
Dataset file as input. It is a file containing input and target columns.
It must have the same number of input and target columns as for the training input file.
For example, it could be the same dataset file used as the training input file.

- **weights file**

this parameter is a field required!
It is a file generated by the model during training phase. It contains the resulting network topology as stored at the end of a training session. Usually this file should not be edited or modified by users, just to preserve its content as generated by the model itself.

- **GPU or CPU**

It is a file generated by the model during training phase. It contains the resulting network topology as stored at the end of a training session. Usually this file should not be edited or modified by users, just to preserve its content as generated by the model itself.
Accepted entries are:

0: serial type execution (on CPU)
1: parallel type execution (on GPU)

If left empty, its default is 1 (GPU)

- **input nodes**

this parameter is a field required!
It is the number of neurons at the first (input) layer of the network. It must exactly correspond to the number of input columns in the dataset input file (Training File field), except the target columns.

- **hidden layers**

It is the number of hidden layers of the MLP network. It can assume only two values (1 or 2). As suggestion this should be selected according the complexity of the problem (usually 1 layer would be sufficient).
If left empty, its default is 1

- **1st hidden layer nodes**

this parameter is a field required!
It is the number of neurons of the first hidden layer of the network. As suggestion this should be selected in a range between a minimum of 2N + 1, where N is the number of input nodes.

- **1st activation function**

It is the choice of which activation function should be associated to neurons of the 1st hidden layer. After this choice, all neurons of the layer will use the same activation function type. The options are:

0: sigmoid
1: threshold
2: linear
3: hyperbolic tangent

By default, the hyperbolic tangent function is used.

- **2nd hidden layer nodes**

It is the number of neurons of the second hidden layer of the network. As suggestion this should be selected smaller than the previous layer.
By default the second hidden layer is empty (not used)

- **2nd activation function**

It is the choice of which activation function should be associated to neurons of the 2nd hidden layer. After this choice, all neurons of the layer will use the same activation function type. The options are:

0: sigmoid
1: threshold
2: linear
3: hyperbolic tangent

By default, if the 2nd layer is activated, the hyperbolic tangent function is used.

- **output nodes**

this parameter is a field required!
It is the number of neurons in the output layer of the network. It must correspond to the number of target columns as contained in the dataset input file (filed Training File). **It is mandatory to set 2 neurons at least.**

- **output activation function**

It is the choice of which activation function should be associated to neurons of the output layer. After this choice, all neurons of the layer will use the same activation function type. The options are:

0: sigmoid
1: threshold
2: linear
3: hyperbolic tangent

By default, the hyperbolic tangent function is used.

3.7 Run Use case

In the use case named "**Run**", the software provides the possibility to run the FMLPGA. The user will be able to use already trained and tested FMLPGA models, their weight configurations, to execute the normal experiments on new datasets.

In the experiment configuration there is also the Help button, redirecting to a web page dedicated to support the user with deep information about all parameters and their default values.

We remark that all parameters labeled by an asterisk are considered required. In all other cases the fields can be left empty (default values are used).

3.7.1 Regression with FMLPGA – Run Parameter Specifications

In the case of Regression_FMLPGA with Run use case, the help page is at the address:
http://dame.dsf.unina.it/FMLPGA_help.html#regr_run

- **input dataset**

this parameter is a field required!
Dataset file as input. It is a file containing input and target columns.

It must have the same number of input and target columns as for the training input file.
For example, it could be the same dataset file used as the training input file.

- **weights file**

this parameter is a field required!
It is a file generated by the model during training phase. It contains the resulting network topology as stored at the end of a training session. Usually this file should not be edited or modified by users, just to preserve its content as generated by the model itself.

- **GPU or CPU**

It is a file generated by the model during training phase. It contains the resulting network topology as stored at the end of a training session. Usually this file should not be edited or modified by users, just to preserve its content as generated by the model itself.
Accepted entries are:

0: serial type execution (on CPU)
1: parallel type execution (on GPU)

If left empty, its default is 1 (GPU)

- **input nodes**

this parameter is a field required!
It is the number of neurons at the first (input) layer of the network. It must exactly correspond to the number of input columns in the dataset input file (Training File field), except the target columns.

- **hidden layers**

It is the number of hidden layers of the MLP network. It can assume only two values (1 or 2). As suggestion this should be selected according the complexity of the problem (usually 1 layer would be sufficient).
If left empty, its default is 1

- **1st hidden layer nodes**

this parameter is a field required!
It is the number of neurons of the first hidden layer of the network. As suggestion this should be selected in a range between a minimum of 2N + 1, where N is the number of input nodes.

- **1st activation function**

It is the choice of which activation function should be associated to neurons of the 1st hidden layer. After this choice, all neurons of the layer will use the same activation function type. The options are:

0: sigmoid
1: threshold
2: linear
3: hyperbolic tangent

By default, the hyperbolic tangent function is used.

- **2nd hidden layer nodes**

It is the number of neurons of the second hidden layer of the network. As suggestion this should be selected smaller than the previous layer.
By default the second hidden layer is empty (not used)

- **2nd activation function**

It is the choice of which activation function should be associated to neurons of the 2nd hidden layer. After this choice, all neurons of the layer will use the same activation function type. The options are:

0: sigmoid
1: threshold
2: linear
3: hyperbolic tangent

By default, if the 2nd layer is activated, the hyperbolic tangent function is used.

- **output activation function**

It is the choice of which activation function should be associated to neurons of the output layer. After this choice, all neurons of the layer will use the same activation function type. The options are:

0: sigmoid
1: threshold
2: linear
3: hyperbolic tangent

By default, the hyperbolic tangent function is used.

3.7.2 Classification with FMLPGA – Run Parameter Specifications

In the case of Classification_FMLPGA with Run use case, the help page is at the address:
http://dame.dsf.unina.it/FMLPGA_help.html#class_run

- **input dataset**

this parameter is a field required!
Dataset file as input. It is a file containing input and target columns.
It must have the same number of input and target columns as for the training input file.
For example, it could be the same dataset file used as the training input file.

- **weights file**

this parameter is a field required!
It is a file generated by the model during training phase. It contains the resulting network topology as stored at the end of a training session. Usually this file should not be edited or modified by users, just to preserve its content as generated by the model itself.

- **GPU or CPU**

It is a file generated by the model during training phase. It contains the resulting network topology as stored at the end of a training session. Usually this file should not be edited or modified by users, just to preserve its content as generated by the model itself.
Accepted entries are:

0: serial type execution (on CPU)
1: parallel type execution (on GPU)

If left empty, its default is 1 (GPU)

- **input nodes**

this parameter is a field required!
It is the number of neurons at the first (input) layer of the network. It must exactly correspond to the number of input columns in the dataset input file (Training File field), except the target columns.

- **hidden layers**

It is the number of hidden layers of the MLP network. It can assume only two values (1 or 2). As suggestion this should be selected according the complexity of the problem (usually 1 layer would be sufficient).
If left empty, its default is 1

- **1st hidden layer nodes**

this parameter is a field required!
It is the number of neurons of the first hidden layer of the network. As suggestion this should be selected in a range between a minimum of 2N + 1, where N is the number of input nodes.

- **1st activation function**

It is the choice of which activation function should be associated to neurons of the 1st hidden layer. After this choice, all neurons of the layer will use the same activation function type. The options are:

0: sigmoid
1: threshold
2: linear
3: hyperbolic tangent

By default, the hyperbolic tangent function is used.

- **2nd hidden layer nodes**

It is the number of neurons of the second hidden layer of the network. As suggestion this should be selected smaller than the previous layer.
By default the second hidden layer is empty (not used)

- **2nd activation function**

It is the choice of which activation function should be associated to neurons of the 2nd hidden layer. After this choice, all neurons of the layer will use the same activation function type. The options are:

0: sigmoid
1: threshold
2: linear
3: hyperbolic tangent

By default, if the 2nd layer is activated, the hyperbolic tangent function is used.

- **output nodes**

this parameter is a field required!
It is the number of neurons in the output layer of the network. It must correspond to the number of target columns as contained in the dataset input file (filed Training File).

- **output activation function**

It is the choice of which activation function should be associated to neurons of the output layer. After this choice, all neurons of the layer will use the same activation function type. The options are:

0: sigmoid
1: threshold
2: linear
3: hyperbolic tangent

By default, the hyperbolic tangent function is used.

4 Examples

This section is dedicated to show some practical examples of the correct use of the web application.
Not all aspects and available options are reported, but a significant sample of features useful for beginners of DAME suite and with a poor experience about data mining methodologies with machine learning algorithms.
In order to do so, very simple and trivial problems will be described.
Further complex examples will be integrated here in the next releases of the documentation.

4.1 Regression XOR problem

The problem can be stated as follows: we want to train a model to learn the logical XOR function between two binary variables. As known, the XOR problem is not a linearly separable problem, so we require to obtain a neural network able to learn to identify the right output value of the XOR function, having a BoK made by possible combinations of two input variable and related correct output.
This is a very trivial problem and in principle it should not be needed any machine learning method. But as remarked, the scope is not to obtain a scientific benefit, but to make practice with the web application.
Let say, it is an example comparable with the classical "print <Hello World> on standard output" implementation problem for beginners in C language.

As first case, we will use the FMLPGA model associated to the regression functionality.

The starting point is to create a new workspace, named **FMLPGAExp** and to populate it by uploading two files:

- **xor.csv**: CSV dataset file for training and test use cases;
- **xor_run.csv**: CSV dataset file for run use case;

Their content description is already described in section 3 of this document.

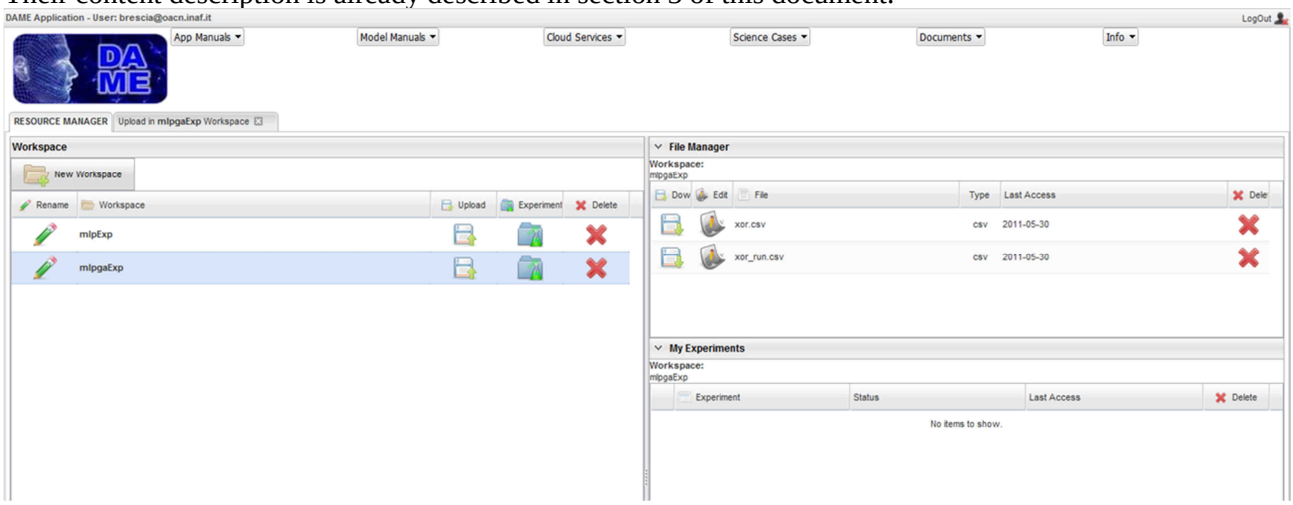


Fig. 18 – The starting point, with a Workspace (FMLPGAExp) created and two data files uploaded

4.1.1 Regression FMLPGA – Train use case

Let suppose we create an experiment named **xorTrain** and we want to configure it. After creation, the new configuration tab is open. Here we select **Regression_FMLPGA** as couple functionality-model of the current experiment and we select also **Train** as use case.

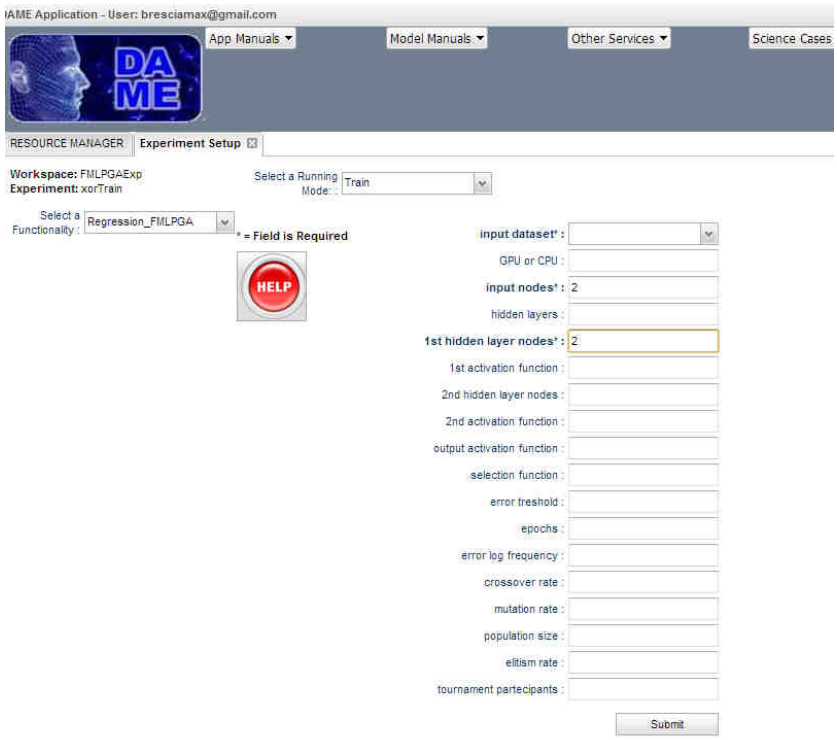


Fig. 19 – The xorTrain experiment configuration tab

Now we have to configure parameters for the experiment. In particular, we will leave empty the not required fields (labels without asterisk).
The meaning of the parameters for this use case are described in section 3.1.1 of this document. As alternative, you can click on the Help button to obtain detailed parameter description and their default values directly from the webapp.
We give xor.csv as training dataset, specifying:

- **Number of input nodes**: 2, because 2 are the input columns in the file;
- **Number of hidden nodes** (first level): 2, as minimal number of hidden nodes (no particularly complex network brain is required to solve the XOR problem). Anyway, we suggest to try with different numbers of such nodes, by gradually incrementing them, to see what happens in terms of training error and convergence speed;
- **Number of output nodes**: 1, because the third column in the input file is the target (correct output for input patterns);

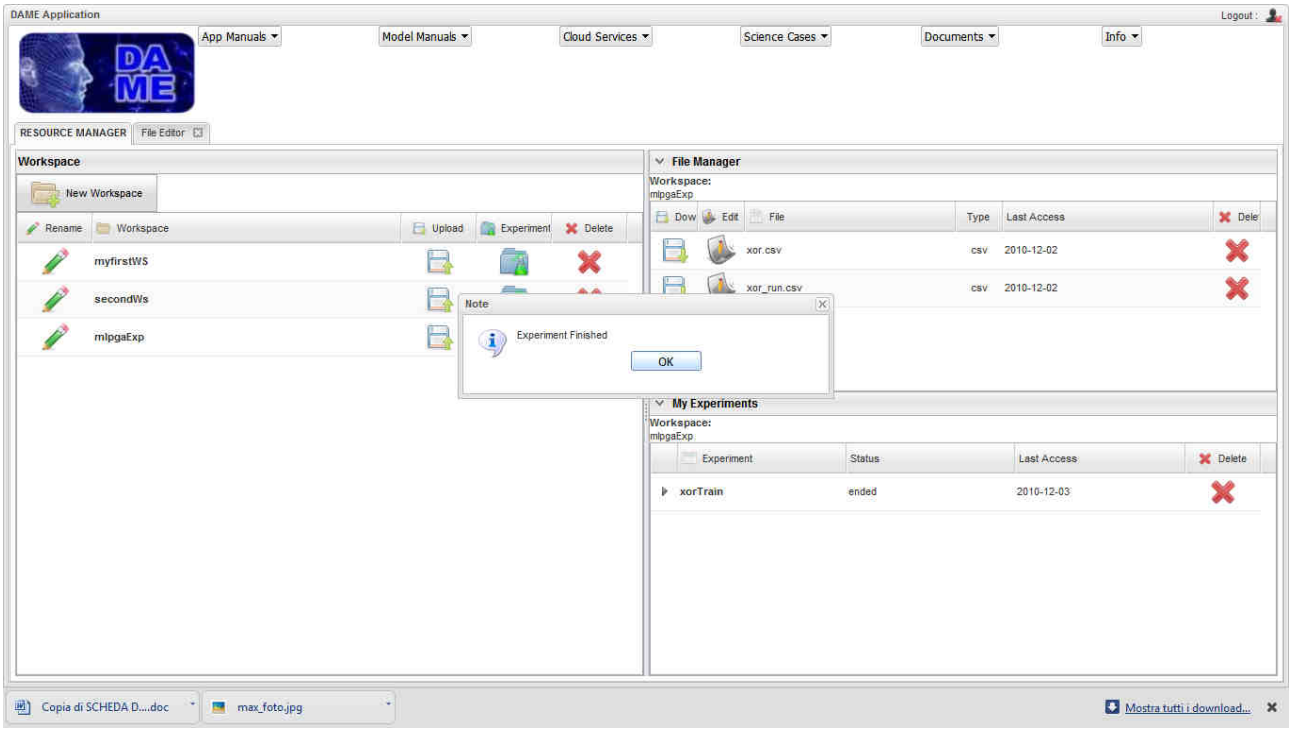


Fig. 20 – The xorTrain experiment status after submission

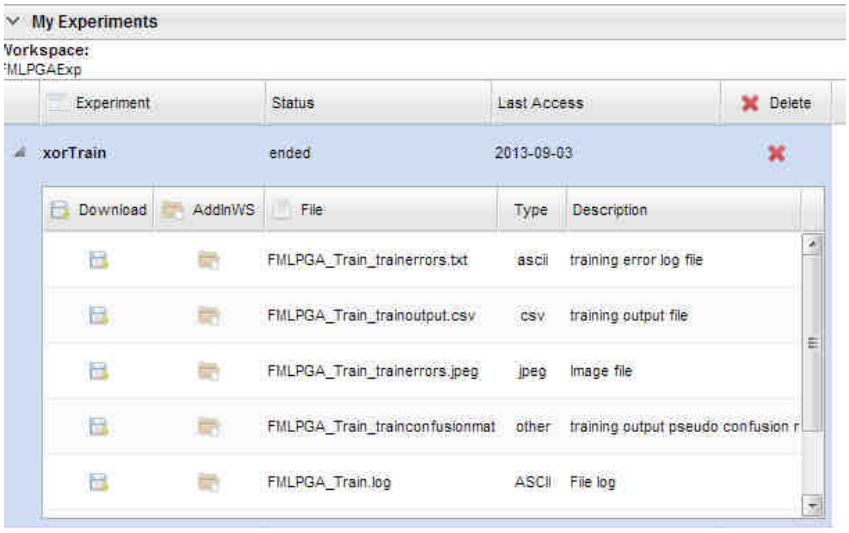


Fig. 21 – The xorTrain experiment output files

The content of output files, obtained at the end of the experiment (available when the status is "ended") is shown in the following (note that in the training phase the file **train_out** is not much relevant). The file **error** reports the training error after a set of iterations indicated in the first column (the error is the MSE of the difference between network output and the target values).

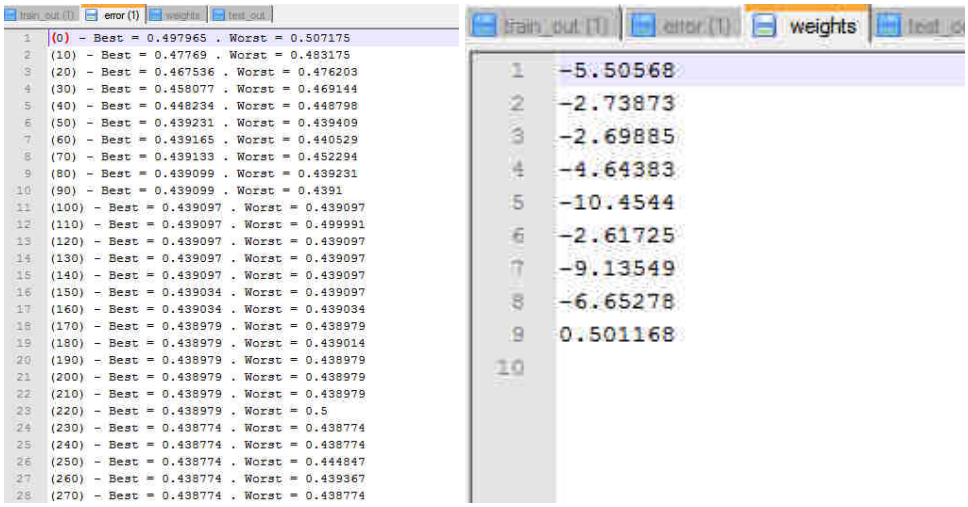


Fig. 22 – The files error (left) and weights (right) output of the xorTrain experiment

The file **weights** has one column and 9 rows. These values are the weights of the connections between the network layers:

- Connections between input and hidden nodes: having 2 input nodes and 2 hidden nodes, we have 4 connections, plus 2 “bias” values for each of the two hidden nodes (first 6 values in the file);
- Connections between hidden and output nodes: having 2 hidden nodes and 1 output node, we have 2 connections, plus the “bias” value for the output node (last 3 values in the file);

4.1.2 Regression FMLPGA – Test use case

The file **weights** can be copied into the input file area (**File Manager**) of the workspace, in order to be re-used in future experiments (for example in this case the test use case). This is because it represents the stored brain of the network, trained to calculate the XOR function.

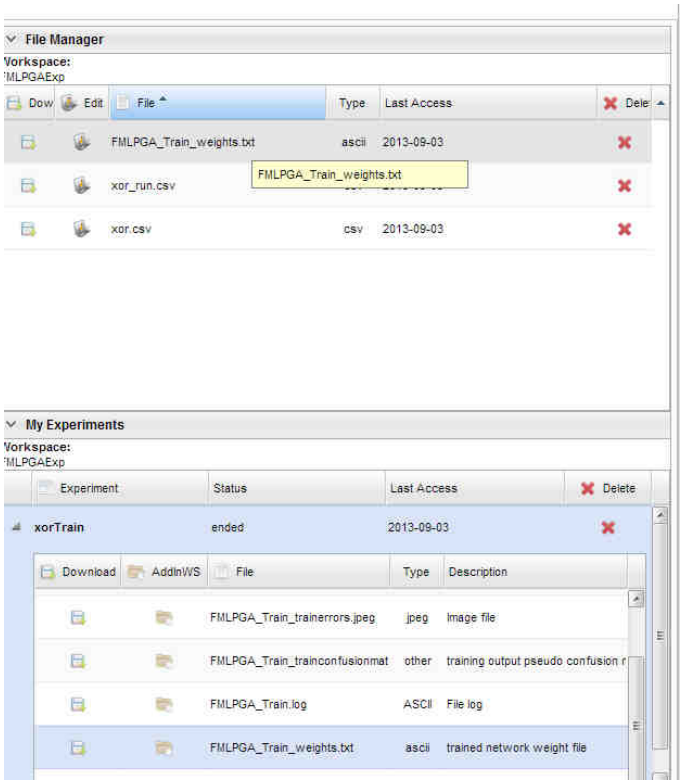


Fig. 23 – The file "weights" copied in the WS input file area for next purposes

So far, we proceed to create a new experiment, named **xorTest**, to verify the training of the network. For simplicity we will re-use the same input dataset (file xor.csv) but in general, the user could use another dataset, uploaded from scratch or extracted from the original training dataset, through file editing options.

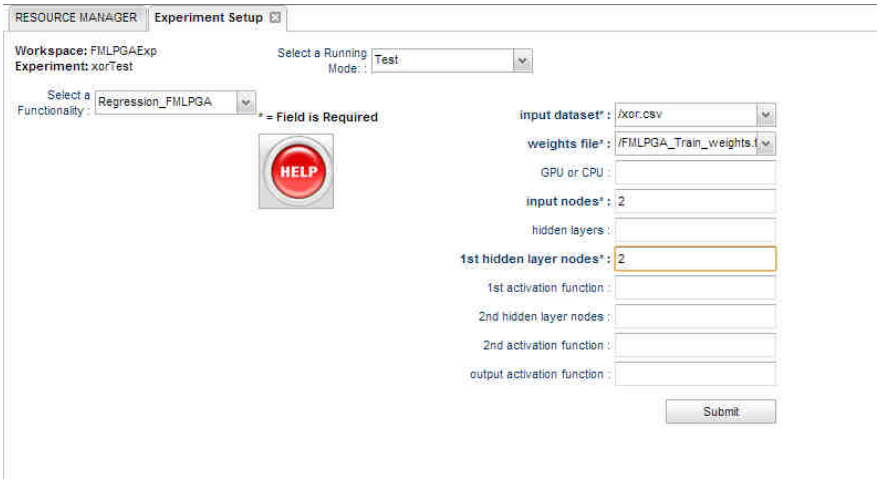


Fig. 24 – The xorTest experiment configuration tab (note "weights" file inserted)

After execution, the experiment **xorTest** will show the output files available.

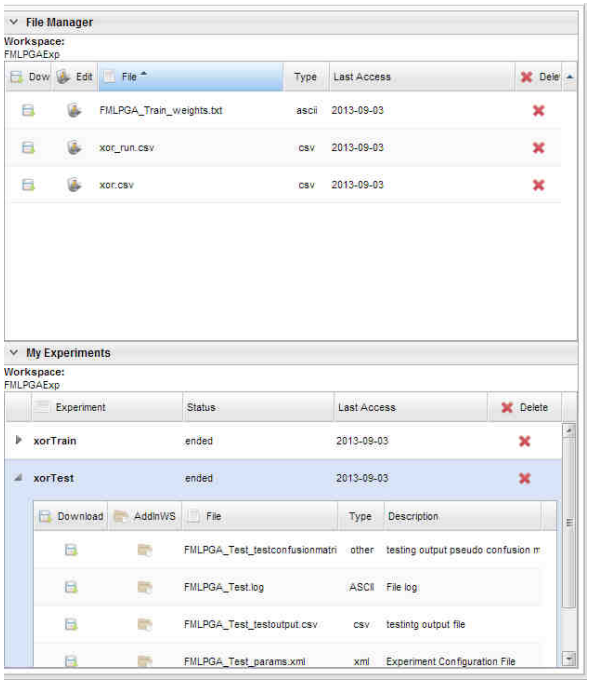


Fig. 25 – The xorTest experiment output files

5 Appendix – References and Acronyms

Abbreviations & Acronyms

A & A	Meaning	A & A	Meaning
AI	Artificial Intelligence	HW	Hardware
ANN	Artificial Neural Network	KDD	Knowledge Discovery in Databases
ARFF	Attribute Relation File Format	IEEE	Institute of Electrical and Electronic Engineers
ASCII	American Standard Code for Information Interchange	INAF	Istituto Nazionale di Astrofisica
BoK	Base of Knowledge	JPEG	Joint Photographic Experts Group
BP	Back Propagation	LAR	Layered Application Architecture
BLL	Business Logic Layer	MDS	Massive Data Sets
CE	Cross Entropy	MLP	Multi Layer Perceptron
CSV	Comma Separated Values	MSE	Mean Square Error
DAL	Data Access Layer	NN	Neural Network
DAME	DAta Mining & Exploration	OAC	Osservatorio Astronomico di Capodimonte
DAPL	Data Access & Process Layer	PC	Personal Computer
DL	Data Layer	PI	Principal Investigator
DM	Data Mining	REDB	Registry & Database
DMM	Data Mining Model	RIA	Rich Internet Application
DMS	Data Mining Suite	SDSS	Sloan Digital Sky Survey
FITS	Flexible Image Transport System	SL	Service Layer
FL	Frontend Layer	SW	Software
FW	FrameWork	UI	User Interface
GPU	Graphical Processing Unit	URI	Uniform Resource Indicator
GRID	Global Resource Information Database	VO	Virtual Observatory
GUI	Graphical User Interface	XML	eXtensible Markup Language

Tab. 3 – Abbreviations and acronyms

Reference & Applicable Documents

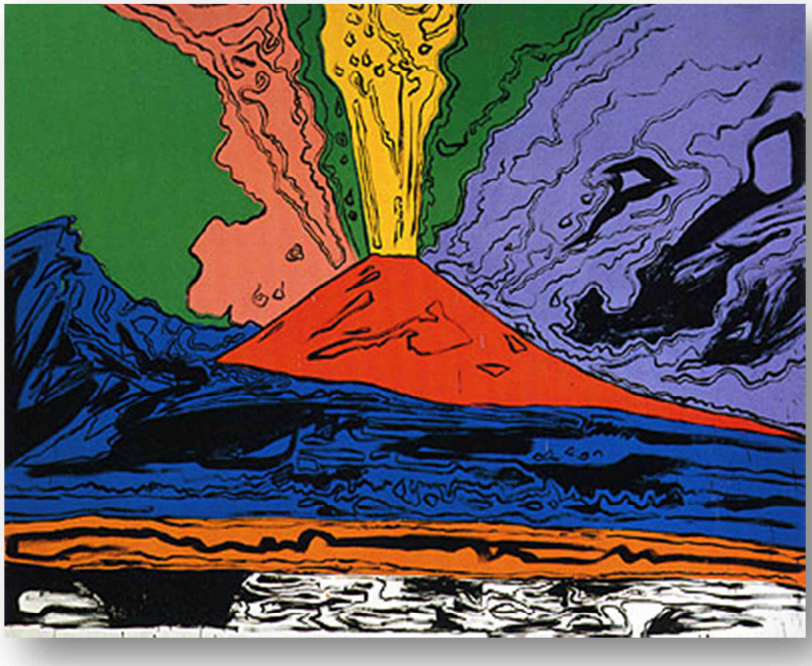
ID	Title / Code	Author	Date
R1	"The Use of Multiple Measurements in Taxonomic Problems", in Annals of Eugenics, 7, p. 179–188	Ronald Fisher	1936
R2	*Neural Networks for Pattern Recognition*. Oxford University Press, GB	Bishop, C. M.	1995
R3	*Neural Computation*	Bishop, C. M., Svensen, M. & Williams, C. K. I.	1998
R4	Data Mining Introductory and Advanced Topics, Prentice-Hall	Dunham, M.	2002
R5	Mining the SDSS archive I. Photometric Redshifts in the Nearby Universe. *Astrophysical Journal*, Vol. 663, pp. 752-764	D'Abrusco, R. et al.	2007
R6	*The Fourth Paradigm*. Microsoft research, Redmond Washington, USA	Hey, T. et al.	2009
R7	Artificial Intelligence, A modern Approach. Second ed. (Prentice Hall)	Russell, S., Norvig, P.	2003
R8	Pattern Classification, A Wiley-Interscience Publication, New York: Wiley	Duda, R.O., Hart, P.E., Stork, D.G.	2001
R9	Neural Networks – A comprehensive Foundation, Second Edition, Prentice Hall	Haykin, S.,	1999
R10	*A practical application of simulated annealing to clustering.* Pattern Recognition 25(4): 401-412	Donald E. Brown D.E., Huntley, C. L.:	1991
R11	*Probabilistic connectionist approaches for the design of good communication codes.* Proc. of the IJCNN, Japan	Babu G. P., Murty M. N.	1993
R12	*Approximations by superpositions of sigmoidal functions.* Mathematics of Control, Signals, and Systems, 2:303–314, no. 4 pp. 303-314	Cybenko, G.	1989
R13	Genetic Algorithm Modeling with GPU Parallel Computing Technology. Neural Nets and Surroundings, Proceedings of 22nd Italian Workshop on Neural Nets, WIRN 2012; Smart Innovation, Systems and Technologies, Vol. 19, Springer http://adsabs.harvard.edu/abs/2012arXiv1211.5481C	Cavuoti et al.	2012
R14	Astrophysical data mining with GPU. A case study: genetic classification of globular clusters. Nuclear Instruments and Methods in Physics Research A, Vol. 720 p. 92–94, Elsevier http://adsabs.harvard.edu/abs/2013arXiv1304.0597C	Cavuoti et al.	2013

Tab. 4 – Reference Documents

ID	Title / Code	Author	Date
A1	SuiteDesign_VONEURAL-PDD-NA-0001-Rel2.0	DAME Working Group	15/10/2008
A2	project_plan_VONEURAL-PLA-NA-0001-Rel2.0	Brescia	19/02/2008
A3	statement_of_work_VONEURAL-SOW-NA-0001-Rel1.0	Brescia	30/05/2007
A4	MLP_user_manual_VONEURAL-MAN-NA-0001-Rel1.0	DAME Working Group	12/10/2007
A5	pipeline_test_VONEURAL-PRO-NA-0001-Rel.1.0	D'Abrusco	17/07/2007
A6	scientific_example_VONEURAL-PRO-NA-0002-Rel.1.1	D'Abrusco/Cavuoti	06/10/2007
A7	frontend_VONEURAL-SDD-NA-0004-Rel1.4	Manna	18/03/2009
A8	FW_VONEURAL-SDD-NA-0005-Rel2.0	Fiore	14/04/2010
A9	REDB_VONEURAL-SDD-NA-0006-Rel1.5	Nocella	29/03/2010
A10	driver_VONEURAL-SDD-NA-0007-Rel0.6	d'Angelo	03/06/2009
A11	dm-model_VONEURAL-SDD-NA-0008-Rel2.0	Cavuoti/Di Guido	22/03/2010
A12	ConfusionMatrixLib_VONEURAL-SPE-NA-0001-Rel1.0	Cavuoti	07/07/2007
A13	softmax_entropy_VONEURAL-SPE-NA-0004-Rel1.0	Skordovski	02/10/2007
A14	VONeuralMLP2.0_VONEURAL-SPE-NA-0007-Rel1.0	Skordovski	20/02/2008
A15	dm_model_VONEURAL-SRS-NA-0005-Rel0.4	Cavuoti	05/01/2009
A16	FANN_MLP_VONEURAL-TRE-NA-0011-Rel1.0	Skordovski, Laurino	30/11/2008
A17	DMPlugins_DAME-TRE-NA-0016-Rel0.3	Di Guido, Brescia	14/04/2010
A18	BetaRelease_ReferenceGuide_DAME-MAN-NA-0009-Rel1.0	Brescia	28/10/2010
A19	BetaRelease_GUI_UserManual_DAME-MAN-NA-0010-Rel1.0	Brescia	03/12/2010

Tab. 5 – Applicable Documents

__oOo__



DAME Program
"we make science discovery happen"