A short description of the Graph Neural Network toolbox

Franco Scarselli
Dipartimento di Ingegneria dell Informazione
Universitá di Siena, via Roma 56, Siena, Italy
email: franco at ing.unisi.it

July 21, 2011

Abstract

The Graph Neural Network (GNN) is a new connectionist model for graph processing. The GNN model is particularly suited for problems whose domains can be represented by a set of patterns and relationships between them. This document describes the Graph Neural Network simulator (version 1.1), which has been developed by me and Grabriele Monfardini.

Contents

1 introduction

This document describes the Graph Neural Network toolbox (version 1.1). The Graph Neural Network (GNN) is a new connectionist model for graph processing. The GNN model is particularly suited for problems whose domains can be represented by a set of patterns and relationships between them. Intuitively, GNNs extend common static neural networks for the information adopted in order to classify (or, more generally, to compute an output for) a pattern. A static network classifies a pattern using only the pattern features, whereas a GNN can use also the related patterns, the connected relationships and, more generally, all the information contained in the domain. A description of the GNN model, its properties and examples of applications can be found in [6, 5].

The GNN toolbox is supposed to work with any version of Matlab after release 5 (and probably even with previous releases). The toolbox is available on line [2].

2 How to install the toolbox

The toolbox installation is straightforward. Decompress the file containing the toolbox anywhere in your computer. Then, add the toolbox main directory and its subdirectories to the Matlab path.

3 A very simple example

The main commands provided by the toolbox are easily illustrated by using the simple experiment displayed in Algorithm 1.

```
1 global dynamicSystem, learning, dataSet, testing;
  makeXXXX;
1 configure GNN.config;
  learn;
  plotTrainingResults;
  test;
```

Algorithm 1: A simple example. Here, **makeXXXX** stands for any benchmark construction procedure in the directory **datasets**.

The meaning of each line of the experiment is as follow.

1. *adding the working variables to the workspace*
   The toolbox stores the working environment into four global variables: `dynamicSystem, learning, dataSet, testing`. More precisely, `dataSet` contains the considered dataset, including the training, the validation and the test set; `dynamicSystem` and `learning` store the learning environment; `testing` includes all the results generated by the test phase. The first command of Algorithm 1 adds those variables to the workspace, so that the user can inspect the working environment. Such a command can be omitted, when the user is not interested in those variables.

2. *dataset construction*
   The sub–directory `datasets` contains a number of scripts, in the form of `makeXXXX`,

that allow to construct some predefined benchmarks. For each synthetic benchmark a configuration file, called `makeXXXX.config`, is included, which defines the parameters of the construction procedure.

3. *configuration*
   The model and the learning environment is configured by the procedure `configure`. By default, `configure` exploits the parameters in the file `GNN.config`.

4. *learning*
   The model is trained by the procedure `learn`.

5. *plotting the history of learning*
   The evolution of some variables during learning can be plotted by the procedure `plotTrainingResults`.

6. *testing*
   The model is tested by the procedure `tested`.

Figure 1 shows the expected data flow and clarifies which variables are used by the procedures. Experiments can be customized by modifying either the configuration files and/or the commands. Actually, a user who needs to rebuild the experiments will only modify the configuration files. On the other hand, to apply GNNs on other datasets, the user must replace `makeXXXX` with his/her own procedure. Some users may also want to add a post processing procedure after `test` in order to compute error measures which are not available in the toolbox.

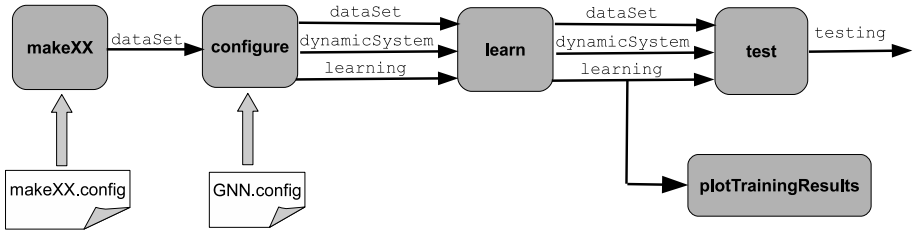

Figure 1: The data flow

More details on the toolbox procedures and variables are disclosed in the following sections.

4 Dataset construction

The data structure `dataSet` contains all the information required by the toolbox to carry out the learning and the test phases. Thus, in order to face a new application, it is sufficient to construct the appropriate `dataSet`.

The variable `dataSet` is a hierarchical data structure which contains mandatory toolbox components and specific problem components. The former components are needed by the toolbox for training and testing; other specific problem components may be introduced by the user to keep track of the parameters used to generate the benchmark, to allow the computation of specific error measures or to store any other information required by the considered application. For having examples of `dataSet`, the user can run the benchmarks provided along with the toolbox. In the following, the mandatory components are described.

The variable `dataSet` must have four substructures: `config`, which includes meta data about the considered problem, and `trainSet`, `validationSet`, `testSet`, which contain the train set, the validation set and the test set, respectively.

4.1 `dataSet.config`

More precisely, `dataSet.config` includes:

- `type` (mandatory)
  The type of the problem, which can be "classification" or "regression" according to whether the faced problem is a classification or a regression application, respectively.

- `rejectLowerThreshold, rejectUpperThreshold` (mandatory in classification problems)
  Two thresholds that specify the reject region and are used to evaluate the GNN model in classification problems. More precisely, in classification problems, a GNN output is considered positive if it is larger than `rejectUpperThreshold` and negative if it is lower that `rejectLowerThreshold`. A value in between the thresholds is rejected: the GNN decision is considered uncertain. Then, the GNN decision is compared with the sign of the target in order to compute the accuracy.

4.2 `dataSet.trainSet, dataSet.validationSet, dataSet.testSet`

The substructures `dataSet.trainSet` `dataSet.validationSet` and `dataSet.testSet` contain the graphs and the targets of the training, the validation and the test set, respectively. In the following, we will assume that each set contains just one graph. However such a fact is not a limit in practice, because a set of graphs can be always merged into a single graph having disconnected components. Since in the GNN framework, unconnected nodes do not effect each others, processing a set of graphs or the corresponding union graph does not make any difference.

The three structures have the following substructures:

- `connMatrix` (mandatory, sparse)
  It is a $N \times N$ matrix, where N is the total number of nodes in the dataset. The matrix, which should be sparse to limit memory allocation, defines the dependences between the nodes. Let $c_{i,j}$ be the (i, j) element of `connMatrix`. Then, $c_{i,j} = 1$ if the i-th node depends on the j-th node and $0 = c_{i,j}$, otherwise. A more clear definition can be obtained observing that, according to the GNN model [6], the state $\boldsymbol{x}_n$ of a node n is computed as
  $$\boldsymbol{x}_n = f_{\boldsymbol{w}}(\boldsymbol{l}_n, \boldsymbol{x}_{\mathrm{ch}[n]}, \boldsymbol{l}_{\mathrm{ch}[n]}, \boldsymbol{l}_{(n,\mathrm{ch}[n])})$$
  where $\boldsymbol{x}_{\mathrm{ch}[n]}$ and $\boldsymbol{l}_{\mathrm{ch}[n]}$ are the states and the labels of the nodes in the neighborhood ch[n] of n, respectively, and $\boldsymbol{l}_{(n,\mathrm{ch}[n])}$ collects the labels of the edges connected to n and $f_{\boldsymbol{w}}$ is the local transition function. Thus, `connMatrix` defines the neighborhoods (the connections): $c_{i,j} = 1$ holds if the j-th node is in the neighborhood of i-th node ((j, i) is in the connections of i) and $c_{i,j} = 0$, otherwise[1].

[1]Notice that, here, `connMatrix` is deliberatively specified using the dependencies between nodes and not the arcs of input graph. Actually, in the GNN processing paradigm, arcs represent dependencies and it may

- `nodeLabels` (mandatory, possibly sparse)
  An $L \times N$ matrix, where L is the label dimension, N is the total number of nodes. The matrix contains the node labels (i.e., $\mathbf{l}_N$ of Eq. (2) in [6]): the i-th column is the label of the i-th node.

- `edgeLabels` (mandatory for graph with edge labels, possibly sparse)
  An $H \times E$ matrix, where H is the dimension of the edge labels and E is the total number of edges in the dataset. The matrix contains the edge labels: the i-th column contains the label of the i-the edge. In order to define the i assigned to each edge, the edges $h \rightarrow k$ are sorted first by the parent h and then, for those edges having the same father, by the child k. An equivalent definition can be obtained by enumerating the edges encountered in a visit of the matrix `maskMatrix` according to the following schema: first the first column is scanned top-down, then the second column is scanned top-down, and so on. In practice, such an edge ordering can be obtained by the Matlab command `[c,r,v]=find(connMatrix)` that stores the sorted indexes of the children and parents in `c` and `r`, respectively: thus, the i-the edge is `(c(i),r(i))`.

- `targets` (mandatory, possibly sparse)
  An $O \times N$ matrix, where O is the output dimension, N is the total number of nodes. The matrix contains the targets: the i-th column is the target of the i-th node.

- `maskMatrix` (mandatory, sparse)
  An $N \times N$ matrix, where N is the total number of nodes. The matrix, which should be sparse to limit memory allocation, is used in squared error function to assign a weighted importance to each GNN output. More precisely, the mse error function is computed as $((\texttt{targets} - \mathbf{o})'\texttt{maskMatrix}(\texttt{targets} - \mathbf{o})$, where $\mathbf{o}$ is the vector of the GNN outputs. In the most obvious case, `maskMatrix` is just a diagonal matrix, where the i-th element d_i of the diagonal is 1 ($d_i = 1$) if the i-th node output is supervised and it is 0 ($d_i = 0$), otherwise. Thus, while, for implementation reasons, the GNN toolbox computes the output of each node, some outputs can be discarded from the error function using `maskMatrix`. More generally, if `maskMatrix` is diagonal, a customized weight can be assigned to each node by an appropriate choice of the d_i: in fact, the mse error function is $\sum_i d_i e_i^2$, where e_i is the difference between the target and the output at the i-th node.

5 Model configuration and learning

The toolbox implements the general GNN models described in [6, 5] with two limits: the positional version of the GNN have not been implemented; the transition net of the linear (non positional) model does not take in input the child and the edge labels, but it can process

happen that `connMatrix` is just the transposed connection matrix of the processed graph. However, such a direct transformation from the input graph to `connMatrix` is correct only if an arc $j \rightarrow i$ expresses the fact that the i-th node depends on the j-th and the converse dependence is not necessarily true. In general, three different cases may happen corresponding to difference meaning of the edges of a graph. A directed edge $j \rightarrow i$ that denotes the dependence of i on j, but not the converse, can be represented by setting $c_{i,j} = 1$ and $c_{j,i} = 0$. An undirected edge $i \leftrightarrow j$ that denotes a mutual and commutative dependence between the nodes i, j can be represented by setting to $c_{i,j} = c_{j,i} = 1$. Finally, a directed edge $j \rightarrow i$ that denotes the presence of two different kinds of dependences between i, j (i.e., the way i depends on j is different from the way j depends on i) can be represented setting $c_{i,j} = c_{j,i} = 1$ so as for undirected graphs, but edge labels must be used to distinguish between the two kinds of dependences.

only the parent label. However, those limits correspond to what specified in the experimental part of the papers. Thus, by running the paper experiments with the same parameters, the same results are expected.

The learning procedure used by the toolbox is specified in [6]: see Table 1 for a short description and Appendix 1 for the implementation details. The two commands `configure` and `learn` allow to initialize the needed data structures and to run the learning procedure. In the following, the two commands are described along with the GNN configuration and the two variables `dynamicSystem, learning`, which contain the information related to the model and the learning environment, respectively.

5.1 `configure`

The command `configure` loads the description of the model and the learning parameters from a configuration file and it constructs the required data structures for the following procedures. The users can provide in input to `configure` his/her own configuration file, f.i., with "`configure XXX.config`". If the configuration file is omitted, the default file `GNN.config` is loaded.

5.2 `learn`

The command `learn` runs the learning procedure. Learning will last a predefined number of epochs as specified by the configuration file. When a validation set is used, the procedure will store the parameters of the network achieving the best performance on the validation set, but it will not stop the procedure before having completed all the epochs.

At a frequency defined by the configuration file, the learning procedure evaluates the current network on the validation set and displays:

- The current epoch number.
- The current error on train set.
- The current error on validation set. The presence of the word "Best" indicates that the best validation error has been achieved.
- The difference between the 1–norm of the Jacobian matrix[2] and the threshold for Jacobian control as defined by the configuration file. Such a difference, which is displayed only if it is larger than 0, allows to verify that the Jacobian control is working correctly. The displayed value should be small and close to 0 or, more precisely, the sum of this value and the Jacobian threshold should be smaller than 1.

Learning can be restarted several times by repeating the command `learn`. Every time `learn` is called, the training will last for the predefined number of epochs. Moreover, the learning procedure can also be stopped by pressing `ctrl+C`. In case of stopping, the training is halted at the exact moment when the interruption key is pressed, so that the last epoch may have been only partially completed. However, since each epoch modifies only slightly the weights, the fact that an epoch is only partially completed is not expected to affect significantly the result of the whole training procedure.

[2]The 1–norm of a matrix $\{a_{i,j}\}$ is defined as $\max_j \sum_i |a_{i,j}|$.

5.3 `The file GNN.config`

The configuration file `GNN.config` contains the parameters of the GNN model and the learning procedure. Those parameters are loaded, by the command `configure`, into the `config` substructure of variables `dynamicSystem` and `learning`. Most of the parameters can be also changed during learning: it is sufficient update them in the variables. The following sections describe the available parameters and display the range of the suggested/admitted values, the default value and the storing location.

5.3.1 Model parameters

- `type` {linear,neural} [neural] `dynamicSystem`
  The type of the GNN. According to a previous terminology, neural stands for the non–linear non–positional GNN, linear for linear non-positional GNN.

- `nStates` (0,inf) [2] `dynamicSystem`
  The state dimension, i.e., the dimension of the state attached to each node.

- `transitionNet, outNet, forcingNet`
  The parameters of the layered networks exploited by the GNN. The network `forcingNet` is used only in linear non–positional GNNs. All those networks have the following parameters.

  - `nLayers` {1,2} [2] `dynamicSystem`
    The number of layers of the network: 2 stands for one hidden layer, 1 for no hidden layer.
  - `outActivationType` {linear,tanh} [tanh] `dynamicSystem`
    The activation function of the output layer, which can be the linear function (linear) or the hyperbolic tangent function (tanh). Notice that if the network has a hidden layer, then the hidden layer neurons always exploit a hyperbolic tangent function. Moreover, in linear non positional GNNs, the transition net must use the hyperbolic tangent function in output layer (see [6]).
  - `nHiddens` (1,inf) [5] `dynamicSystem`
    Number of hidden neurons.
  - `weightRange` (10^{-10},1) [0.01] `dynamicSystem`
    The network weights are initialized by a uniform random function in the range [-weightRange,weightRange].

- *Jacobian control parameters*
  When the Jacobian control is enabled, a penalty term is added to the error function in order to ensure that the global dynamic system is contractive [6]. In practice, the Jacobian control is not used by linear GNNs and it has to be used in non–linear GNNs. The following parameters specify the Jacobian control model.

  - `useJacobianControl` {0,1} [1] `dynamicSystem`
    This parameter enables or disable the Jacobian control.
  - `jacobianThreshold` $(0, 1 - 10^{-9})$ [0.85] `dynamicSystem`
    It is the threshold ν described in the point NON–LINEAR (NON–POSITIONAL)

GNN in [6]: the Jacobian penalty is larger than 0, when the 1-norm of the Jacobian matrix is over such a threshold.

- `jacobianFactorCoeff` $(0,\infty)$ [1000] `dynamicSystem`
  It is the coefficient β described in the point NON–LINEAR (NON–POSITIONAL) GNN in [6]: the coefficient allows to weight the importance of the mse error w.r.t. the penalty term.

- *Saturation control parameters*
  The saturation control aims to avoid that neurons becomes very saturated[3]. Such a control is implemented by adding a decay factor to the error function, when the output of the node is larger than a predefined threshold. Saturation control is used only by the non–linear GNNs and in hidden neurons.

  - `useSaturationControl` {0,1} [1] `dynamicSystem`
    This parameter enables and disables the saturation control.
  - `saturationThreshold` $[0, 1 - 10^{-9}]$ [0.99] `dynamicSystem`
    The threshold for saturation control: saturation control is active when the node output is larger than such a threshold.
  - `saturationCoeff` $(0,\infty)$ [0.001] `dynamicSystem`
    The coefficient of the saturation control: it allows to weight the importance of the mse error w.r.t. the forgetting factor of the saturation control.tance of the mse error w.r.t. the forgetting factor of the saturation control.

- `useLabelledEdges` {0,1} [0] `dynamicSystem`
  It enables and disables the use of the labeled edges. If `dataSet.trainSet` and, as a consequence, `dataSet.testSet` and `dataSet.trainSet` contain a structure `edgeLabel`, then `useLabelledEdges` must be 1 and it must be 0, otherwise.

- `errorFunction` {mse,quadratic,outmult,ranking,autoassociator} [mse] `dynamicSystem`
  The error function adopted by the learning procedure. The error can be: the common mean squared error (mse); the quadratic error adopted in the half hot example in [5] (quadratic); the quadratic error with output multiplication adopted in web page ranking example in [6] (outmult); the logarithm error used in [3] (ranking); the output network is an autoassociator and the error function for autoassociators (autoassociator).

5.3.2 Learning algorithm parameters

- `learningSteps` $(1,\infty)$ [500] `learning`
  The number of learning epochs.

- *The parameters specifying the number of iterations of the forward and backward procedures*
  As described in the Algorithm of Figure 1 in [6], the stable state is computed by the forward procedure, which iteratively updates the node states, and the gradient is computed by the backward procedure, which iteratively backpropagates the gradient. The forward procedure repeats the state computation up to when the relative difference between two

[3]Node saturation is a problem, because when the neuron becomes saturated, the gradient becomes close to 0: in the limit case, the gradient can be exactly 0 due to numerical rounding.

consecutive states $\frac{\mathbf{x}(t)-\mathbf{x}(t-1)}{\mathbf{x}(t)}$ is less than a given threshold `forwardStopCoefficient` or a maximum number of repetitions `maxForwardSteps` is reached. Similarly, the backward procedure repeats the backpropagation up to when the delta $\mathbf{z}(t)$ is smaller than a given threshold `backwardStopCoefficient` or a maximum number of iterations `maxBackwardSteps` is reached.

  - `maxForwardSteps` (1,∞) [10] `learning`
    Maximum number of iterations for the forward procedure.
  - `maxBackwardSteps` (1,∞) [10] `learning`
    Maximum number of iterations for the backward procedure.
  - `forwardStopCoefficient` $(10^{-15}, 10^{-2})$ $[10^{-8}]$ `learning`
    Stop threshold for the forward procedure.
  - `backwardStopCoefficient` $(10^{-15}, 10^{-2})$ $[10^{-8}]$ `learning`
    Stop threshold for the backward procedure.

- *Validation parameters*
  At a predefined frequency, the current GNN model is evaluated on validation set. The GNN state on validation is computed using the forward procedure of the algorithm in Table 1 of [6].

  - `useValidation` 0,1 [1] `dynamicSystem`
    It enables or disable the use of the validation set.
  - `stepsForValidation` (1,50) [20] `dynamicSystem`
    It is the frequency at which the GNN is evaluated on the validation set. With the same frequency, the learning procedure shows the error on training set, the error on validation and the Jacobian error.
  - `maxStepsForValidation` (1,∞) [10] `dynamicSystem`
    The forward procedure on validation set is iterated until the relative difference $\frac{\mathbf{x}(t)-\mathbf{x}(t-1)}{\mathbf{x}(t)}$ between two consecutive states is less than `stopCoefficientForValidation`.
  - `stepsForValidation` $(10^{-15}, 10^{-2})$ $[10^{-5}]$ `dynamicSystem`
    The iteration of the forward procedure on validation set is also stopped if the maximum number of iterations `stepsForValidation` is reached.
  - `useValidationMistakenPatterns` 0,1 [0] `dynamicSystem`
    Such a parameter allows to select the error function for the the validation set. Actually, if the parameter is set to 0, the error function adopted for the validation set is the same used on training set. Otherwise, if the parameter is set to 0, the error is given by the number of mistaken patterns, possible balanced according to the relative number of positive and negative classes (see later): such a setting can be useful in classification problems.

- *Resilient parameters*
  The GNN learning algorithm adopts resilient backpropagation . The default parameters of resilient are those adopted in the original paper [4].

  - `deltaMax` (50,200) [100] `learning.rProp`
    It is denoted Δ_{max} in [4].

- `deltaMin` $(10^{-10}, 10^{-2})$ [10^{-7}] `learning.rProp`
  It is denoted Δ_{max} in [4].
- `etaP` (1.1,1.5) [1.2] `learning.rProp`
  It is denoted η^+ in [4].
- `etaM` (0.2,0.9) [0.5] `learning.rProp`
  It is denoted η^- in [4].

5.3.3 Other parameters

- *Log to file parameters*
  The toolbox can log to file.
  - `useLogFile` 0,1 [1] `dynamicSystem`
    When this parameters is enabled, the toolbox save to a log file having the same name as the config file and with a `.log` extension.
  - `useAutoSave` 0,1 [1] `dynamicSystem`
    When this parameter is set to 1, the toolbox periodically saves the whole learning environment to a file, having the same name as the config file and with a `.mat` extension. Pay attention, saving the learning environment may make the training slow.
  - `autoSaveEveryTotValidations` (1,∞) [5] `dynamicSystem`
    When autosaving is enabled, this parameter specifies the frequency in epochs of the saving.
- *History saving parameters*
  The toolbox stores into the structure `learning` the evolution of some learning variables. In order to limit the consumed memory, the storing can be disabled.
  - `saveErrorHistory` 0,1 [1] `dynamicSystem`
    It enables or disables the storing of the error history.
  - `saveJacobianHistory` 0,1 [1] `dynamicSystem`
    It enables or disables the storing of the Jacobian error history.
  - `saveSaturationHistory` 0,1 [1] `dynamicSystem`
    It enables or disables the storing of the saturation error history.
  - `saveStabilityCoefficientHistory` 0,1 [1] `dynamicSystem`
    It enables or disables the storing of the history of stability coefficient.
  - `saveIterationHistory` 0,1 [1] `dynamicSystem`
    It enables or disables the storing of the history of the numbers of forward and backward iterations.
  - `saveStateHistory` 0,1 [0] `dynamicSystem`
    It enables or disables the storing of the history of states.

5.4 `dynamicSystem`

The variable `dynamicSystem` includes data about the current GNN model and encoding network. Such a data is modified by the learning procedure (not by the testing procedure).

The variable contains the data at the interruption time, if the learning procedure is stopped, either because the maximum number of epochs has been reached or because the user pressed the Matlab interruption key. Notice that historical information (f.i., the parameters of the GNN achieved the best result on validation set) are not stored in `dynamicSystem` but in the variable `learning`.

In the following the most important substructures of `dynamicSystem` are listed and explained.

- `config`
  It includes several configuration parameters of the GNN model and the learning environment. The toolbox read those parameters from the configuration file `GNN.config`, which have been described in the previous section.

- `state`
  A $S \times N$ matrix, where S is the dimension of the state and N is the total number of nodes in the dataset. The matrix contains the global state on train set (i.e., $\mathbf{x}(t)$ of Eq. (4) in [6]). Notice that `state` equals to the stable state (i.e., $\mathbf{x}$ of Eq. (2) in [6]) after the feedforward phase.

- `parameters`
  A variable that contains the current GNN parameters. The parameters are stored into three different substructures `outNet`, `transitionNet`, `forcingNet`, according to the network they belong. Actually, a GNN models is made by three networks that implement an out function, a transition function and a forcing function, where the last is used only for the GNN linear model. The parameters of each network are stored into four substructures:

  - `weights1`
    A matrix $a \times b$ containing the first layer weights: if the network has a hidden layer then a is the number of hidden weights, otherwise, if the network has no hidden layer, a is the number of outputs: in both cases b is the number of inputs neurons.
  - `weights2`
    A matrix $a \times b$ containing the second layer weights, where a is the number of outputs and b the number of hidden neurons. The substructure is set only if the network has a hidden layer.
  - `bias1`
    A vector containing the first layer biases. More precisely, it contains the biases of the hidden layer, if the network has a hidden layer, or the biases of the output, otherwise.
  - `bias2`
    A vector containing the second layer biases. The vector is set only if the network has a hidden layer, when the second layer is the output layer.

5.5 `learning`

The variable `learning` includes data about the learning environment. Such a data is modified by learning procedure (not by testing procedure). The variable has three substructures: `config`, `current`, `history`.

- `config`
  It includes several configuration parameters of the learning environment. The toolbox read those parameters from the configuration file `GNN.config`, which have been described in the previous section.

- `current`
  It contains the working parameters of the learning environment.

- `history`
  It contains the history of some learning variables including:

  - `trainErrorHistory`
    The train error history.
  - `validationErrorHistory`
    The validation error history.
  - `forwardItHistory`
    The number of iterations of the forward procedure.
  - `backwardItHistory`
    The number of iterations of the backward procedure.
  - `jacobianHistory`
    The history of the Jacobian error.
  - `stabilityCoefficientHistory`
    The history of the stability coefficient. The stability coefficient is $\frac{\mathbf{x}(t+1)-\mathbf{x}(t)}{\mathbf{x}(t)}$ computed at the last the iteration (of each epoch) of the forward procedure. A large stability coefficient may suggest that the Jacobian control is not working and the forward procedure is not capable of making the system converge.
  - `saturationCoefficient`
    The history of the saturation coefficient. The saturation coefficient is the mean output level of the hidden neurons.

5.6 `testing`

The variable `testing` includes the results of the testing procedure. The data is slitted into two substructures `current`, `optimal`, containing the results achieved by the GNN at the end of the learning (current) and those achieved by the GNN that got the best error on the validation set (optimal). The data is further splitted into three substructures `trainSet`, `validationSet`, `testSet` with the results of the corresponding sets. Finally, each one of those structures includes the following substructures.

- `error`
  The error computed by the error function adopted for training, f.i., mse by default.

- `accuracy`
  The accuracy.

- `out`
  The GNN output.

- `mistakenPatternIndex` (only in classification problems)
  The indexes of the nodes that have been misclassified, i.e., whose output is wrong.

- `mistakenPatterns` (only in classification problems)
  The labels of the the nodes that have been misclassified, i.e., whose output is wrong.

- `mistakenTargets` (only in classification problems)
  The targets of the node that have been misclassified, i.e., whose output is wrong.

- `maxError`
  The maximum error, over all the nodes. The error is the modulus of the difference between the GNN output and the target.

- `maxRelativeError`
  The maximum relative error, over all the nodes. The relative error is the modulus of the difference between the GNN output and the target divided by the target.

- `acc5percent`
  The accuracy over all the nodes, when the correctly classified nodes are those for which the relative error is less than 5%.

- `acc5percent`
  The accuracy over all the nodes, when the correctly classified nodes are those for which the relative error is less than 10%.

Notice that a user can exploit the above substructures to implement a post processing and to compute his/her own performance measure.

6 The benchmarks

The toolbox contains a number of benchmarks including the datasets used for the papers [6, 5]. Those datasets can be rebuilt using the files makeXXXX in the directory "datasets". For each dataset, there is a configuration file with the same name as the command and ".config" extension.

- `makeCliqueDataset`
  The clique example in [5].

- `makeHalfHotDataset`
  The half hot on uniform graphs example in [5].

- `makeMutagenicDataset`
  The mutagenesis example in [6].

- `makeNeighborsDataset`
  The neighbors example in [5].

- `makeOddEvenDataset`
  A benchmark where the goal is to decide, for each node, whether the integer label is even or odd. In this benchmark, the connectivity of the graph is not useful.

- `makeParityDataset`
  The parity example in [1].
- `makeSecondOrderNeighborsDataset`
  The second order Neighbors example in [5].
- `makeSubGraphDataset`
  The subgraph matching example in [6].
- `makeTreeDepthDataset`
  The tree depth example in [5].
- `makeWebPagesScoringDataset`
  The web page ranking example in [6].
- `makeGeneralDataset`
  A procedure to load generic datasets.

References

[1] M. Gori, G. Monfardini, and F. Scarselli. A new model for learning in graph domains. In *Neural Networks, 2005. IJCNN '05. Proceedings. 2005 IEEE International Joint Conference on*, volume 2, pages 729 – 734 vol. 2, july-4 aug. 2005.

[2] G. Monfardini and F. Scarselli. The graph neural networks toolbox: it is available at *http://airgroup.dii.unisi.it/projects/GraphNeuralNetwork/download.htm*, 2004.

[3] A. Pucci, M. Gori, M. Hagenbuchner, F. Scarselli, and AC Tsoi. Investigation into the application of graph neural networks to large-scale recommender systems. *Systems Science*, 32(4):17–26, 2006.

[4] M. Riedmiller and H. Braun. A direct adaptive method for faster backpropagation learning: The RPROP algorithm. In *Neural Networks, 1993., IEEE International Conference on*, pages 586–591. IEEE, 2002.

[5] F. Scarselli, M. Gori, G. Monfardini, Ah Chung Tsoi, and M. Hagenbuchner. Computational capabilities of graph neural networks. *IEEE Transactions on Neural Networks*, 20(1):81–102, 2009.

[6] F. Scarselli, M. Gori, G. Monfardini, Ah Chung Tsoi, and M. Hagenbuchner. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.