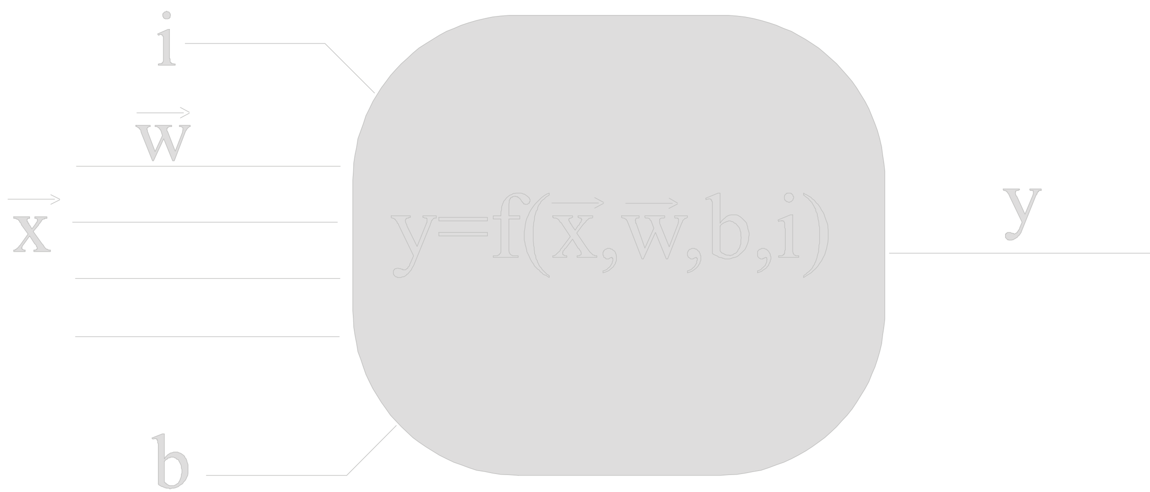
# Neural Networks Library

# In Java

Manual V 5.2



# INDEX

Part 1 Basic Concepts ........................................ 1
1. Introduction ........................................ 1
2. The Library ........................................ 3
3. Definition script language ........................................ 4
4. Why Java ........................................ 5
5. Internal structure ........................................ 6
Part 2 Using the Library ........................................ 8
6. Network definition ........................................ 8
7. Forward computing ........................................ 9
8. Learning ........................................ 11
9. Other NN manipulations ........................................ 13
10. NN parameters ........................................ 14
11. Get and Set methods ........................................ 14

# Part 1
# Basic Concepts

## 1. Introduction

Some 12 years of growth in the study and use of neural networks, has produced a boundless quantity of papers and books. But let me briefly introduce here the basic concepts. For more information you can read the recognised book [1], or later,[2],[3].

The inspiring analogy was the biological nervous system. But pretty soon it became a real new computational paradigm. Actually, the original first appearance was the Mc Culloch and Pitts[4] model of neuron (1943), the Hebbian[5] rule of synapses modification (1949) and particularly the Rosenblatt's[6] "perceptron" (1958). But the Minsky's and Papert[7] critical analysis of these simple models (1969) buried this subject for a long time until the "Multi-layer perceptron" and the Error Back Propagation method from McClelland and Rumelhart[1] (1989).

The biological neuron is the basic element which every animal, more sophisticated than a bacterium, uses to manage the interaction with its environment. Therefore it must perform a sort of computation to set out the action following a sensory input. This kind of computation is parallel because the huge quantity of neurons linked together. It is fail safe because it is distributed and redundant, and fuzzy because it is analogical. i.e. it is very attractive!

If the biological neuron can be a little complicated, particularly if you pretend to simulate every physical and chemical aspect, its usual model in neural networks (short for Artificial Neuronal Networks) is very simplified.

Substantially, a neuron performs a computation of input values (other neurons output) weighted by the synapses strength. The result of this computation is outputted to other neurons. Usually, the computation task is simulated by a squash (i.e. non linear) function applied to the weighted sum of inputs. Recently more attention has been paid to the dynamic of neuron activity that is impulsive. But usually, neural networks are made by neuron models that perform an integration of these spikes.

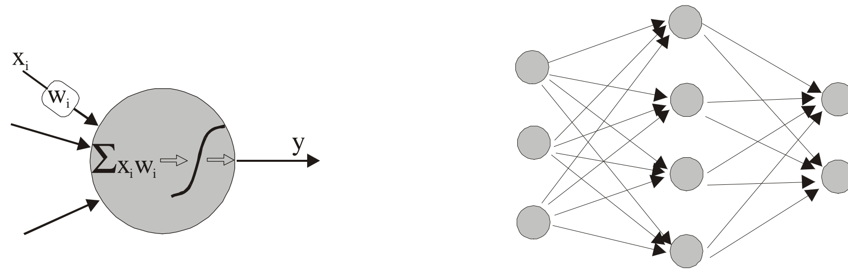


Figure 1 : Neuron model and Feed-Forward Neural Network

The Error Back Propagation (EBP) learning method, described in 1989, has produced a great interest in the Feed-Forward neural networks. This kind of neural network (NN for short) is a multi-layered "perceptron" structure, where each layer is fully linked with the next (i.e. without recursion). In fact, neural networks are historically thought of as a learnable structure. But until EBP, there wasn't an efficient and practical learning method. EBP is a supervised learning method. It means that the NN is trained by a list of input-output pair values. In this case the NN becomes a sort of multivariate non-linear regression model. In a Feed-Forward NN there is an input layer (or simple buffer), an output layer and one or more "hidden" (i.e. intermediate) layers. It has been demonstrated that a NN with 2 hidden layers and an appropriate number of nodes can approximate every function.

Feed-Forward NN with EBP learning method are a very multi-purpose system. They can be seen as a statistical method[10], a non-linear controller, a filter, an agent behaviour system and every other complex input-output function approximation and generalization[11][12].

Actually, NN are utilized in dynamic systems too. But in this case, research is still in progress. However there is a simplified NN with feedback loop that is successfully used with EBP learning method. This kind of NN with memory is called NN with context memory or Elman[8] memory. The context memory it is a buffer where the output activity of a layer is copied. This buffer is linked with the same layer (or another one) and holds its value for the next cycle of a feed-forward NN. Practically, it is a semi-recursive system.

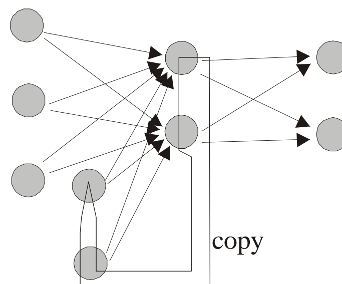



Figure 2 NN with context memory

Feed-Forward NN with EBP learning method is not the only configuration used and studied. For example, Radial Basis Functions (RBF) are another universal approximation system and the Self Organized Maps (SOM)[9] are an efficient clustering method.

At the same time other learning methods are being investigated and used: from supervised to self-organizing, and particularly evolution methods with populations of NN.

Different structures can use different kinds of computational nodes, and Feed-Forward NN can use different nodes as well. For this reason NN represent a general paradigm of computational nodes networks more than a single structure model.

# 2. The Library

If you are looking for a software structure to implement a Feed-Forward NN with 3 layers (this is the usual EBP NN), the simplest solution is a couple of matrix where you can store the weights. One for input-hidden link [i,h], and another for hidden-output link [h,o]. A fixed kind of computational node (linear) for the input and a fixed kind of computational node (sigmoid) for hidden and output layer. And, of course, 3 arrays where you can store the output value for each node.

But if you would like to implement the general distributed computational paradigm that NN represents, you need something more flexible. In fact, the aim of this library is to implement **a general frame of computational nodes**.

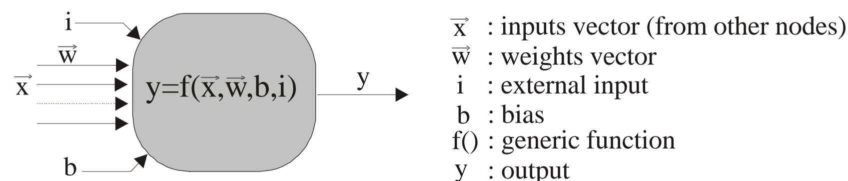



Figure 3 Node

The sole constraint in this model is the layers organization. This means that the nodes must cluster in layers. Each layer has a sequential number. This number marks the order of computation. The computation function of the NN starts from layer 0 (the input layer) and continues in sequence to the last layer.

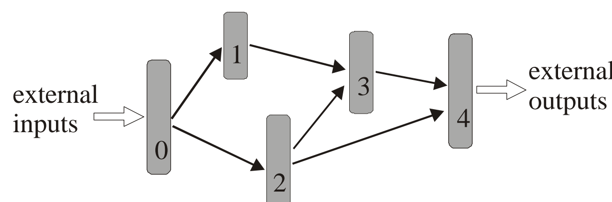



Figure 4 Layers structure as computational sequence

There is no limit for the number of layers and each layer can have any number of nodes (but same type in this release). In theory it is possible to assign one layer for every node. One layer can link every other layer. Besides, it is possible to specialize links for sub-clusters of nodes inside a layer. For instance it is possible to have a link for nodes 0 to 5 of layer 2 with nodes 3 to 6 of layer 3. i.e. one node can be linked with any other node.

It is possible to define a memory context too. Actually, it is possible to define several buffers. A memory buffer is a layer like the others and has a sequence number. The buffer is filled (using the external node input) by the output of another layer, for which it is the memory. If its sequence number is less than the buffered layer it will be computed in the next net computation call and it acts as a memory context. Otherwise it will be computed in the same cycle and could simulate lateral (inter-layer) links or other purposes.

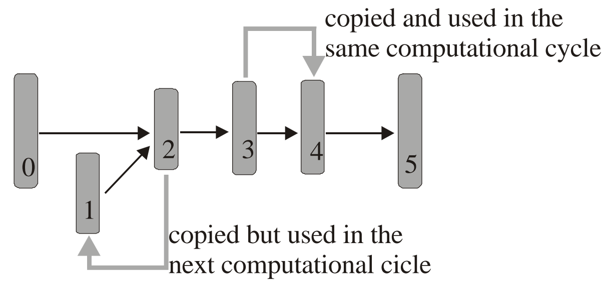



Figure 5 Memory buffers

In conclusion, it is possible to define structures that are very complicated and bizarre as well as simple 3-layers NN (also the recursive link is not prevented). Finally, it is possible to define 2-dimensional layers. This kind of layer holds neighbour pointers (left, right, up, down) inside each node for topological uses on the node computation phase.

# 3. Definition script language

A simple script language is provided for new NN definition. It contains 3 types of records:

- Layer definition
- Link definition
- Bias definition

They don't need to appear in a particular order and can be mixed, but they must start with its first keyword.

These records can be read inside a Java program as an array of String or from a text file.

**layer**=n [**tnode**=n[,m] **nname**=xxxxx… **copytoml**=n ]

layer=n : n is the sequence layer number

tnode=n[,m] : n is the numbers of nodes . If m!=0 , this layer is 2D layer and m is the y dimension
nname=xxxx… : is the class name of node
copytoml=n : if exists is the layer number buffer for this layer

**linktype**=[xxx…] **fromlayer**=n[(h[,k[,p,q]])] **tolayer**= n[(h[,k[,p,q]])] [**value**=na,[,nb]]

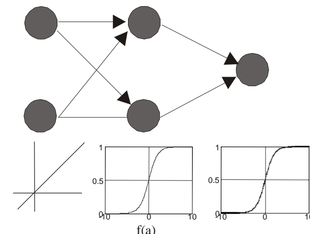
linktype=xxx… : is a string "all" (means all-to-all link) or "one" (one-to-one link)

fromlayer= n[(h[,k[,p,q]])] : n is the layer from and h and k can define a sub-cluster of nodes (from h to k). q,p can define a cluster rectangle in case of 2D layer
tolayer= n[(h[,k[,p,q]])] : same meaning but related to the target layer
value=na[,nb] : na and nb define the range of random value. Only na means fixed value

**biasval**=na[,nb] **oflayer**=n[(h[,k[,p,q]])]

biasval=na[,nb] : na and nb define the range of random value. Only na means fixed value
oflayer=n[(h[,k[,p,q]])] : same meaning of previous record

Example:
layer=0 tnode=2 nname=NodeLin
layer=1 tnode=2 nname=NodeSigm
layer=2 tnode=1 nname=NodeSigm
linktype=all fromlayer=0 tolayer=1
linktype=all fromlayer=1 tolayer=2




# 4. Why Java

This library was part of a European Project for a distributed platform for evolutionary computation (DREAM project)[14]. In this project Java was the language chosen because of its intrinsic mobility (weak mobility) feature. The possibility of transporting instances as standard feature is a must for distributed systems with agents. Here we can summarize the principal features for a NN library in Java:

- Portability (different platforms)
- Mobility (for agents implementation)
- Dynamic binding and reflection (for external node definition)

Portability doesn't need further explanation. Mobility means that a NN with its behaviour (i.e. all of its weights) can be sent to another network node to resume work. The third point allows the making of a frame where computational nodes are defined at the instantiation time[13]. Customers' class nodes can be loaded into the NN frame as well as built-in nodes. They can stay in the user's directory or they can be included into the library. With the classical feature of object-oriented programming, the user can define his node class inheriting pre-existent node classes at different levels.

# 5. Internal structure

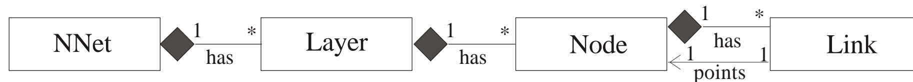



Figure 6 Relationships

| NNet | Layer | *Node* | Link |
|---|---|---|---|
| Layer[]<br>some coefficients<br>and parameeters<br>Class Layer (inner) | Node[]<br>some buffers<br>(for win-take-all) | Link[]<br>inp (ext. input)<br>out (outp. val.)<br>bias is Link[0]<br>err (used in EBP)<br>buffers for out and err<br>(because 2 phases comp.)<br>Node[4] in case of 2D layer<br>Class Link (inner)<br>references to nnet and layer | Node : node from<br>wgt : weight from<br>wgtb: buffer (EBP)<br>if bias Node=null |
| methods for:<br>- **compute**<br>- EBP learn<br>- mutate wgts<br>- save and load<br>- read and modify nodes and links values<br>- … | private methods<br>called by NNet<br>methods; they<br>perform NNet<br>operations localy<br>(Layer is a<br>inner class) | trf() : abstract<br>(forward computation)<br>out(): move outbuffer to out<br>lrn(): abstract<br>init(): weights init<br>mut(): weights mutation<br>err(): move errbuff to err<br>seterr(): for EBP use<br>methods get and set<br>for node variables and links | |

Figure 7 Principal classes

***Forward Phase***
This is the most important method of NNet. It performs the NN computing from layer 0 to the last layer. For each layer every node is computed applying trf() function and the result is loaded on out-buffer. After this first phase, every layer node is again computed applying its out() function. This just moves out-buffer value to out field. This 2-phase procedure allows intra-layer links in theory, but it can be utilized for win-take-all strategy as well. Win-take-all means that just one of the layer nodes is active. After each layer computation, its output values are loaded into its buffer memory layer, if it exists.

***Learning Phase***
In this phase lrn() method is activated. This time layers are scanned in reverse mode, from the last one to the first one. This is because this method must comply with the most used learning method: EBP. Also in this case there are 2 phases for each layer.

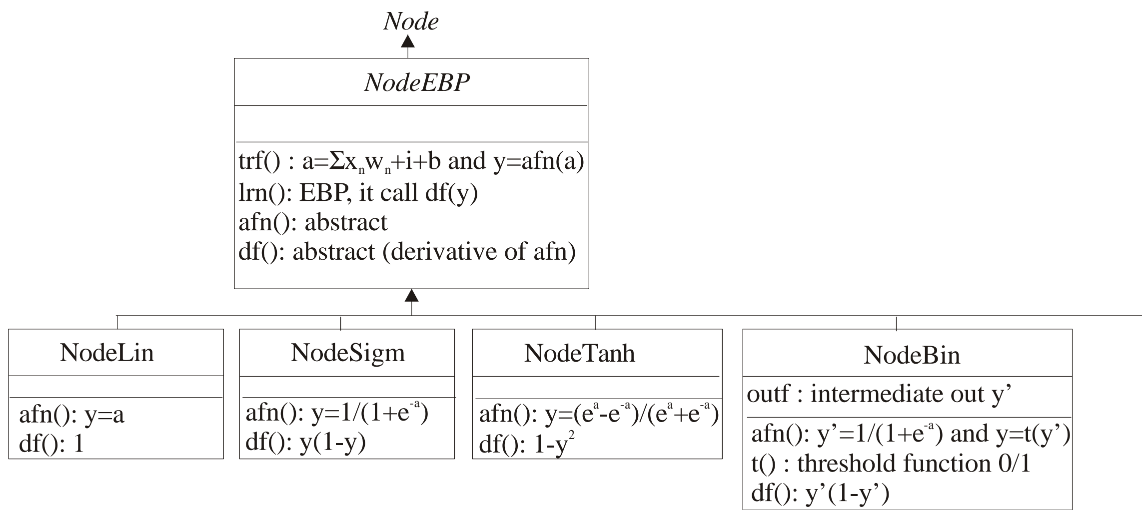



Figure 8 An example of inheritance diagram for nodes

# Part 2
# Using the Library

## 6. Network definition

The user can save or load networks structures by dedicated library methods and constructor.

- saveNNet(…) (on file or print stream) , toString() (on string)
- NNet(…) (constructor from file or string)

The toString() method, is the implementation of standard Java toString() method, and it saves the complex NN structure.

Whereas, to define a new NN, user can utilize a simplified constructor for simple 3 layered NN, or the script language described in the previous part.

- NNet(n1,n2,n3,flag mem) where n are the number of nodes for each layer
- NNet(…) using a file or an array of strings (records) and the script language

The script language allows to define a layered NN of any complexity. Only one record for each layer must be defined. But it is possible to define any number of link record to specify detailed links.

For instance if you want define the first layer (layer number 0) with 10 nodes of linear type (as usual for the first layer that is substantially a buffer) you have to write:

```
layer=0 tnode=10 nname=NodeLin
```

In fact, NodeLin is the name of linear node (using EBP) built-in the library.

Or if you want define the layer 2 that has layer 1 as its memory buffer, you have to write:

```
layer=2 tnode=6 nname=NodeSigm copytoml=1
```

For link record the syntax can be a little more complicated if you want a detailed link. For instance if you want define a simple single link all-to-all from layer 1 to layer 2 you can simply write:

```
link=all fromlayer=1 tolayer=2
```

In this case all nodes of layer 1 will be linked with all nodes of layer 2 using default range values for random initialising of weights.

But if you want define a link from the first 5 nodes of layer 1 to the first 3 nodes of layer 2, and a link one-to-one from the other 4 nodes of layer 1 to the other 4 nodes of layer 2 with fixed values .7, you have to write:

```
link=all fromlayer=1(0,5) tolayer=2(0,3)
link=one fromlayer=1(6,9) tolayer=2(4,7) values=.7
```

In the same way you can use bias records to detail bias values for the nodes of a layer. If no bias records are used, the biases are initialised using default values (=0).

When a double value for "tnodes" is written in the layer record, that means this layer is a 2-dimensional layer. For example:

```
layer=2 tnodes=10,8 nname=NodeSigm
```

means that layer 2 contains a 10x8 matrix of nodes. In this case each node has the reference to its 4 neighbours.

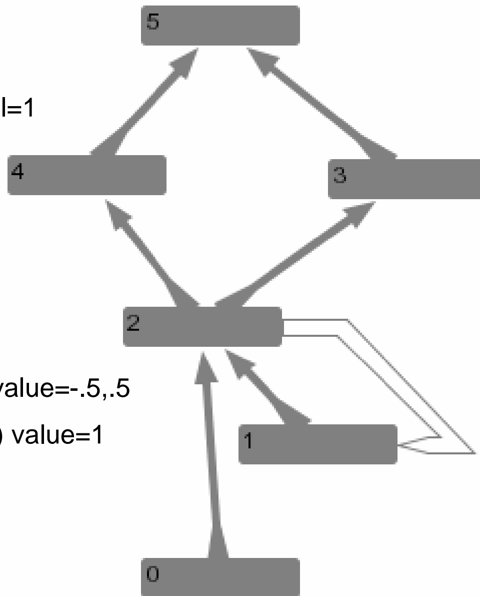
With 2-dimensional layers you can have more detailed link description and you can address a sub-matrix of nodes. For example:

```
link=all fromlayer=2(0,3,0,2) tolayer=3
```

means that you link nodes 0,0 0,1 0,2 0,3 1,0 1,1 1,2 1,3 2,0 2,1 2,2 2,3 with all nodes of layer 3.

Example of a complex structure:

```
layer=0 tnode=10 nname=NodeLin
layer=1 tnode=6 nname=NodeLin
layer=2 tnode=6 nname=NodeSigm copytoml=1
layer=3 tnode=4 nname=NodeSigm
layer=4 tnode=8 nname=NodeSigm
layer=5 tnode=6 nname=NodeSigm
linktype=All fromlayer=2 tolayer=4
linktype=All fromlayer=2 tolayer=3
linktype=All fromlayer=4(0,3) tolayer=5(0,1) value=-.5,.5
linktype=One fromlayer=3(0,3) tolayer=5(2,5) value=1
linktype=All fromlayer=0 tolayer=2
linktype=All fromlayer=1 tolayer=2
biasval=.5 oflayer=5
```




# 7. Forward computing

To compute a NN, in this library, means to compute nodes of each layer in sequence from layer 0 to the last one. The basic method that perform this NN computation is:

- frwNNet() without any parameter

Before calling this method you have to fill the external inp field of nodes (which have external input), and after the computation you have to collect output values of nodes (that are seen as output nodes).

More useful are the methods that allow to define the external input that will be loaded into the first layer, and the external output buffer that will receive the output values of the last layer:

- frwNNet(inp[], out[]) where inp and out are float arrays
- rec-out=frwNNet(rec-inp) where rec-inp is a string and the method returns a string

If you would like to use a NN as a filter to process an input file and producing an output file you can use the method:

- NNetFilter(input-file-name, output-file-name)

This method scans each record of the input file and applies the frwNNet method to it. The output is written on the output file.

Suppose you want test a NN, regarding of its behaviour, using a list of correct inp-out pairs values. In case of single sample, you can use:

- testNNet(inp[],out[]) where inp and out are arrays of input and expected output value
- testNNet(rec) where rec is a string contains the pairs values

Both methods return a float value that represents the mean squared error of this output.

Or in case of testing a list :

- testNNet(rec[] or file-name)

In this case the method returns the mean squared error of all samples.

Another more sophisticated testing method is:

- verifyNNet(input-file-name)

This method reads the pairs values on input file and prints the input values, the NN output values, the correct output for comparison and the squared mean error for each record of input file to the console output.

Actually, the NnetFilter and the verifyNNet are simplified methods that came from a complex and flexible method that allows also the training phase:

- outNNet(…………..) see methods description on technical JavaDoc documentation

When you use a string or a file to provide values, these values (always numeric values) must simply be separated by spaces. For example: 0.12 0.8 .55 1 0.8 could be a record for 5 input values or for a pair of 3 inputs and 2 output or any other combination of input-output. The methods know the NN structure and can load the values properly.

Flow diagram of forward phase:

- for each layer starting from layer 0 to last layer
    - for each node of layer
        - applies frw() node method
    - for each node of layer
        - applies out() node method (moves oub to out and clear outb)
    - if this layer has a memory buffer
        - for each node of layer
            - puts the out value to the inp field (external input) of related (in terms of number) node on buffer
            - stops when reach the last nodes of layer or the last node of buffer layer (if they have different number of nodes)

# 8. Learning

As you have seen in Part 1, learning phase consists in a reversal scanning of layer. From last layer to first layer. This is because learning must include the most important algorithm: EBP. But learning phase is more general and it simply applies the lrn() node method. The kind of learning algorithm is therefore referred to the node implementation.

The basic learning method is:

- lrnNNet()  without any parameter

This method just applies the learning phase. This can generate confusion, because if you want apply the EBP method you have to load the err node field by the derivative of squared error (d err/ d out).

Flow diagram of learning phase (lrn()):

- For each layer starting from the last one to the first one
    - For each node of layer
        - Applies the err() node method (moves errb to err and clear errb)
    - For each node of layer
        - Applies the lrn() node method

As you can see, also in learning  there are 2 phases.

## *Error Back Propagation*

More useful are methods (specialized for EBP) that allow to define the input and output  values for training (i.e. the example that the NN must imitate). In this case the methods apply both, a forward phase and a learning phase :

- ebplrnNNet(inp[],outexp[])  where outexp is now the forced out (out expected)
- epblrnNNet(inp[],out[],outexp[]) where out is a buffer for NN output
- epblrnNNet(rec) where rec is a string containing the pair values inp-out

These methods apply  the input values. Compute the NN with frwNNet() and compare the NN output with the forced output. Load the err field of last layer with the derivative of error and finally apply the learning phase. These methods return the mean squared error related to this inp-out pair.

In case of complete training set you can use:

- ebplrnNNet(rec[] or training-file-name)

This method scans the training file (or the array of strings) end applies the EBP algorithm for each record. This method returns the mean squared error related to the entire training-set.

Another more sophisticated method is:

- trainNNet(input-file-name, output-file-name)

This method read a file where are a list of pairs inp-out. It interprets this values as a training set and applies EBP for each record. Besides, it writes the NN answer on output file (like filter method) and writes the squared error on the console output (for each record). This method is a simplified version of outNNet method (see previous chapter).

As said in the previous chapter, a training set record (or string) is just a list of input-output values separated by spaces. Obviously, they must be in the same number of first layer and last layer nodes. But it is possible to provide a non-value for expected values. In this case the EBP methods interpret this non-value as any-value. In other words, they don't correct this particular output (error=0).

Example (4 input and 2 output): 0.1 0.8 1 0.5 0 *

It is possible to define a non-value output also in array of float. You have just to use the Float.NaN value or the NNet.NaN value.

For EBP methods the general diagram is:

- For each training record (pair values)
  - For each node of first layer
    - Loads input values into inp node field
  - Applies forward phase (frw())
  - For each node of last layer
    - Compares the output value with the expected value
    - Calculate the derivative of sqerr (=out-outexp)
    - Loads errb node field with this value
  - Applies learning phase (lrn())

## *Mutation*

An important and very general learning algorithm is the Evolutionary Algorithm. This is a population based learning system. Some NN methods are provided to simplify the implementation of this algorithm. Basically, weights mutation methods and cross-copy.

Like lrn() method, also weights mutation is transferred to node level. This means that the basic mutation method is implemented in Node class (mut() method). This allows to customize mutation in case of particular nodes like FuzzyAND node for instance.

To apply a weights mutation you can use:

- wgtmutNNet(…..)

Where many parameters make this method very flexible (see methods technical JavaDoc documentation). Or simplified versions:

- wgtmutNNet(dev, flag bias) Gaussian distribution with std deviation = dev
- wgtmutNNet(ra,rb,flag bias) Linear distribution with ra-rb range

These 2 methods apply a modification to all weights with a random excursion.

It is also possible to perform a sort of crossover:

- crossLNNet(with NN, using layer)

Where the first parameter is the other NN which you want exchange the weights, and the second one is the layer that exchanges its link weights. If the two NN have not the same structure the result is unpredictable.

Finally, a method for obtaining a vector of weights and its reverse (for loading a vector into the NN) are provided. This can be intended as a simplified genotype that user can manipulate in more sophisticated way:

- Vector=getwgtNNet(flag bias)
- setwgtNNet(Vector, flag bias)

# 9. Other NN manipulations

To reinitialize the NN you can use:

- inirandNNet(ra,rb,flag bias)

If you want apply a decreasing procedure to all weights you can use:

- wgtdecNNet(coefficient, flag bias)

This method reduces each weight of a proportional quantity (w=w-w*coef). The bias flag decides if bias are included.

To copy one NN to another one (i.e. to override its weights):

- copyWNNet(to NN)

This method copies weights and bias from one NN to another one and if these two NN have different structure the result is unpredictable.

To create a new copy:

- cloneNNet() or clone()

The clone() method is substantially a synonym of cloneNNet() and it is provided as implementation of standard Java object clone() method. In fact it returns a general Object (while cloneNNet() returns a NN instance).

Sometimes you could need to blank the memory buffers. In this case you can use:

- clearAllMemBuff() puts 0 on inp field of every buffer layer
- clearMemBuffOf(layer) just for this buffer layer

# 10. NN parameters

The NN instance holds same parameters as well as the layers structure. These parameters are used for weights initialization, for leaning phase etc.

This parameters are public but set methods are also provided:

- setSeed(num) sets the seed for random generator. If no num, time is the seed (default).
- setRandRange(ra,rb) sets the weights initialization range (def. –0.6 +0.6)
- setFbias(t/f) make bias learnable (in ebp procedure) or not (def. no)
- setGLrnCoef(val) sets a general learning coefficient (def. 0)
- setLearnCoef(val) sets the EBP learning coeff. (def. 0.3)
- setLearnMom(val) sets the EBP momentun coeff. (def. 0.3)

The first two methods are static and can be utilized before the NN creation.

In addition to NN parameters there are also some buffers on each layer. These buffers can be accessed by the node classes to perform some particular function as win-take-all policy for instance. These buffers are: maxval (a float), nodemax (a node pointer), gbuff (a float).

# 11. Get and Set methods

A lot of get and set methods let you to :

- Read the number of layers and the number of nodes for each layer.
- Read the output values of a layer.
- Write the external input on the inp fields of a layer nodes.
- Set and get the bias of a layer nodes.
- Set and get the err fields of a layer nodes.
- Get the pointer of a node of a layer.

See the technical JavaDoc description for more details.

## References


[1] McClelland J.D., Rumelhart D.E., *Parallel Distributed Processing. Explorations in the Microstructure of Cognition* Cambridge Mass., Mit Press 1989

[2] Bishop C.M., *Neural Networks for Pattern Recognition*, Oxford University Press 1996

[3] Haykim S., *Neural Networks: a comprehensive foundation*, Prentice Hall 1999

[4] Mc Culloch,W.S.,and W.Pitts, *A logical calculus of the ideas immanent in nervous activity*, Bulletin of Mathematical Biophisics , 1943

[5] Hebb,D.O., *The Organization of Behavior: A Neuropsychological Theory*, New York: Wiley, 1949

[6] Rosenblatt,F., *The Perceptron: A probabilistic model for information storage and organization in the brain*, Psycological Rewiew 1958

[7] Minsky,M.L.,and S.A.Papert, *Perceptrons*, Cambridge, MA:Mit Press 1969

[8] Elman, J.L. *Finding structure in time.* Cognitive Science **14** 1990 179-211

[9] Kohonen, T. *Associative Memory: A System Theoretic Approach*. Springer-Varlag, Berlin. . 1977

[10] Warren S. Sarle, *Neural Networks and Statistical Models*, Proceedings of the Nineteenth Annual SAS Users Group International Conference, April, 1994,SAS Institute Inc., Cary, NC, USA

[11] Nolfi S.,Floreano D., *Evolutionary Robotics*, MIT Press 2000

[12] Cangelosi A., Parisi D., *Simulating the Evolution of Language*, Springer Verlag London 2001

[13] Horstmann C.S, Cornell G., *Core Java V1,2*,Sun Microsystem Press USA 2001

[14] DREAM group (omissis)