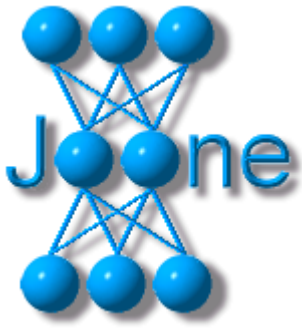


Java Object Oriented Neural Engine

The Complete Guide

All you need to know about Joone

17. January 2007

by Paolo Marrone
http://www.joone.org

Table of Contents

I would like to present the objectives that I had in mind when I started to write the first lines of code of Joone.
My dream was (and still is) to create a framework to implement a new approach the use of neural networks. I felt this necessity because the biggest (and unresolved until now) problem is to find the fittest network for a given problem, without falling into local minima, thus finding the best architecture.

Okay - you'll say - this is what we can do simply by training some randomly initialized neural network with a supervised or unsupervised algorithm.
Yes, it's true, but this is just scholastic theory, because training only one neural network, especially for hard problems of the real life, is not enough.
To find the best neural network is a really hard task because we need to determine many parameters of the net such as the number of the layers, how many neurons for each layer, the transfer function, the value of the learning rate, the momentum, etc... often causing frustrating failures.

The basic idea is to have an environment to easily train many neural networks in parallel, initialised with different weights, parameters, or different architectures, so the user can find the best NN simply by selecting the fittest neural network after the training process.
Not only that, but this process could continue retraining the selected NNs until some final parameter is reached (i.e. a low RMSE value) like a distillation process. The best architecture can be discovered by Joone, not by the user! Many programs today exist that permit selection of the fittest neural network applying a genetic algorithm. I want to go beyond this, because my goal is to build a flexible environment programmable by the end user, so any existing or newly discovered global optimisation algorithm can be implemented. This is why Joone has its own distributed training environment and why it is based on a cloneable engine.
My dreams aren't finished, because another one is to make a trained NN easily usable and distributable by the end user. For example, I'm imagining an assurance company that continuously trains many neural networks on customer's risk evaluation[1] (using the results of historical cases), distributing the best 'distilled' (or genetically evolved) neural network to its sales force, so that they can use it on their mobile devices.
This is why a neural network built with Joone is serializable and remotely transportable using any wired or wireless protocol, and it is easily runnable using a simple, small and generalized program.
Moreover, my dream can become a more solid reality thanks to the advent of handheld devices like mobile phones and PDA having inside a java virtual machine. Joone is ready to run on them, too.

Hoping you'll find our work interesting and useful, thank you for your interest in Joone.

Paolo Marrone
and the Joone team

[1] The ethics (and the law in many countries) forbids to make racial, sexual, religious (and others) discriminations. Consequently, a decisional system based on such personal characteristics **cannot** be built.

1 Introduction

1.1 Intended Audience

This paper describes the technical concepts underlying the core engine of Joone, explaining in detail the architectural design that is at its foundation.
This paper is intended to provide the users - or anyone interested to use Joone - with the knowledge of the basic mechanisms of the core engine, so that anyone can understand how to use it and expand it to resolve one's needs.

A basic knowledge of the basic concepts underlying the artificial neural networks is required, consequently, who doesn't own such a know-how should read some good introductory book on the argument.

1.2 What is Joone

Joone (http://www.joone.org/) is a Java framework to build and run AI applications based on neural networks. Joone applications can be built on a local machine, be trained on a distributed environment and run on whatever device.
Joone consists of a modular architecture based on linkable components that can be extended to build new learning algorithms and neural networks architectures.

All the components have some basic specific features, like persistence, multithreading, serialization and parameterisation. These features guarantee scalability, reliability and expansibility, all mandatory features to reach the final goal to represent the future standard on the AI world.

Joone applications are built out of components. Components are pluggable, reusable, and persistent code modules. Components are written by developers. AI experts and designers can build applications by gluing together components with graphical editors, and controlling the logic with scripts.

Around the components will be based all the modules and applications written with Joone.
Joone can be used to build *Custom Systems*, adopted in an *Embedded* manner to enhance an existing application, or employed to build applications on *Mobile Devices*:

1.2.1 Custom systems

A great need of the industrial market is to have the possibility to resolve business problems suitable with neural networks (or with AI applications in general). Joone wants to represent the optimal solution to build applications to satisfy such needs (i.e. bank loan assessment, sales forecasting, etc.).

Its characteristics are optimal to build custom applications driven from the user's needs, where it's important to have flexibility, scalability and portability.
Each enhancement of Joone will be compatible also with the necessity of build applications more quickly than other products on the market, so Joone can gain a large market share and become one of the most used neural network frameworks.

1.2.2 Embedded systems

Into the core engine, the components are the bricks to build whatever neural network architecture. Their purpose is to create AI applications writing Java code that uses the Joone's API.
In the respect of the goal that aims to obtain a wide adoption of Joone from the market, the license of the core engine is the Lesser General Public License (LGPL), so everyone can freely embed the engine into existing or new applications. **This will never change**.

The business model of Joone contemplates the possibility of provide more components to satisfy the users needs to create several neural network architectures and algorithms, so they can embed Joone into whatever application (i.e. data mining systems, automatic categorization for search engines, customer classification, etc.)

1.2.3 Mobile Devices

One long-term goal of Joone is to become the basic framework to provide a computational engine to AI applications suitable for the mobile devices (phones, PDA, etc.).
The demand for software products available for such kind of devices is growing, therefore in the future a new market of applications to satisfy these needs will be open, gaining the interest of the industrial world.

Joone wants to be present in that market and represent the main framework to distribute and run personal or corporate AI applications (i.e. handwriting and voice recognition, support to the sales force, sales or financial forecasting, etc.).
The core engine of Joone is already suitable for small devices, having a small footprint and being runnable on Personal Java environments.

1.3 About this Guide

This guide is composed by the following chapters (the asterisks indicate the skill required to correctly understand the exposed concepts, as listed at the end of this paragraph):

Chapter 1 – Introduction (*)
This Chapter contains a brief description of Joone, what it is and what are its possible applications in several fields of the professional world.

Chapter 2 – Getting and installing Joone (*)
This is a starter guide to learn how to download and install all the Joone framework and how to obtain a runnable version from the source code.

Chapters 3-7 – Concepts and technical details (**)(***)
These chapters illustrate the basic concepts underlying the core engine. They explain the main features of the core engine from a functional point of view, and, for whose that are interested to the technical implementation, each chapter ends with a paragraph named 'technical details', where a more detailed look about how the described features have been implemented is given.

Chapter 8 – Common Architectures (**)
This is a practical guide about how to build the most common neural network architectures, like the temporal, recurrent, unsupervised and the mixed ones. For each of them an example is built using the visual editor. This Chapter can be intended as a complement of the Editor User Guide, and its goal is to give a first look about some potential applications of Joone. / TO BE COMPLETED /

Chapter 9 – Applying Joone (***)
This Chapter explains the main features of Joone using concrete and useful examples written in java code. Applying the programming techniques described in this chapter everyone can build a custom java application that uses joone as internal neural network engine. / TO BE COMPLETED /

Legend:
* No specific skill required
** Basic knowledge about artificial neural networks
*** Good understanding of UML and Java code

1.4 Acknowledgements

Joone was made possible thanks to the many people that have agreed my initial idea and have extended the initial code adding new ideas, suggestions and, mainly, good and often documented source code. This is the demonstration that also in a complex stuff like Artificial Intelligence, thanks to the Open Source model it's possible to obtain the collaboration of valid and skilled programmers to build a complete, stable and powerful framework.

Paolo Marrone, the founder and project manager of Joone, wants to thank all the guys who have contributed continuously and for a long period of time, writing good java code, and also giving a great support represented by very interesting proposals and suggestions. They are (listed in alphabetical order):

Andrea Corti, Huascar Fiorletta, Harry Glasgow, Boris Jansen, Julien Norman, Paul Sinclair, Thomas Lionel Smets

Thanks also to the following people for their unvaluable contribution:

Gavin Alford, Mark Allen, Jan Boonen, Yan Cheng Cheok, Ka-Hing Cheung, Pascal Deschenes, Jan Erik Garshol, Jack Hawkins, Nathan Hindley, Olivier Hussenet, Shen Linlin, Casey Marshall, Christian Ribeaud, Anat Rozenzon, Trevis Silvers, Jerry R. Vos

Do you want to see your name listed above? Join us: any contribution is **always** welcome, therefore if you have built some new component, new feature, or you have fixed some bug, contact me (pmarrone@users.sourceforge.net) and I'll be very happy to insert your name in the above list of contributors. You can also contact me even if you're not a java developer, but, as expert of neural networks, you'd like to help me to implement some new component or new training algorithm. Of course the Forums on the web site and my email address are always open for any idea or suggestion. Thanks.

I want also to thank all the authors of the following O.S. external packages used by Joone:

- **JhotDraw** http://sourceforge.net/projects/jhotdraw
- **BeanShell** http://sourceforge.net/projects/beanshell
- **jEdit-Syntax** http://sourceforge.net/projects/jedit-syntax
- **Log4J** http://jakarta.apache.org/log4j
- **HSSF-POI** http://jakarta.apache.org/poi
- **L2FProd** http://common.l2fprod.com
- **NachoCalendar** http://nachocalendar.sourceforge.net
- **XStream** http://xstream.codehaus.org
- **VisAD** http://www.ssec.wisc.edu/~billh/visad.html

A particular acknowledgment to:

- SourceForge.net, thanks to which all this has been possible
- **Zero G Software**, Inc. for InstallAnywhere, the multi platform auto installer program used by Joone

2 Getting and Installing Joone

2.1 Platform and requirements

Joone is written in 100% pure Java and can run on whatever platform for which a Java Runtime Environment v. 1.4+ is available.
Due to his direct experience, or because he has received informations from other users, the author can assure the compatibility of Joone with the following operating systems[2]:

- Linux
- Mac OSX
- Windows 2000
- Windows XP
- SUN Solaris

About the memory requirement, it depends on the complexity of the neural network used, but generally the availability of at least 256MB of RAM, even if not mandatory, is strongly recommended.

Due to its small footprint, a minimal version of the Joone's core engine can run also on mobile devices (PDA) running J2ME Personal Profile. The author ran without problems the sample XOR neural network on a HP-Compaq IPAQ device provided with 32MB of flash memory using successfully both Jeode and IBM J9 JVMs.

2.2 Installing the binary distribution

Joone is distributed both in source and compiled form. The compiled distribution (named also the *binary* distribution) is available both for the core engine and the GUI editor. We'll see how to download and install them on your machine.

2.2.1 The Core Engine

[2] InstallAnywhere is a registered trademark of Zero G Software, Inc.
Mac OS is a registered trademark of Apple Computer, Inc.
Solaris and Java are trademarks of Sun Microsystems, Inc.
Windows is a registered trademark of Microsoft Corporation.
All other marks are properties of their respective owners.

The compiled form of the core engine can be useful to run a whatever application written in java that uses the Joone's engine API, as deeply described in the next chapters. All the classes are contained into the library *joone-engine.jar*. This library cannot run stand-alone, as it doesn't contain any *main* class, but it must be put into the classpath of the application that needs to use Joone.
Depending on which Joone engine's packages are used, you need also to put in the classpath some external packages provided in a separate downloadable file.

Here are explained the steps to execute to correctly install the core engine's libraries:

1. Download the core engine's binary distribution file *joone-engine-x.y.z.zip* (where x, y and z are respectively the major/minor version and the build number of the last available distribution)
2. Download *joone-ext.zip*, the file containing the needed external libraries Unzip both the above files into a predefined directory of your file system (we'll name it `<base_dir>`). At this point you should have a directory tree as below (we omitted the unessential files):

```
<base_dir>
      Joone-engine.jar
      ...
      <ext>
            bsh.jar
            crimson.jar
            jakarta-poi.jar
            log4j.jar
            ...
      <samples>
            ...
```

3. Put the *joone-engine.jar* ando also the *<ext>*.jar* files into your classpath
4. Run your own application

Depending on the engine's packages your application uses, you need to put only the needed libraries on your classpath, as depicted in the following table:

Library	**Purpose**	**When used**
joone-engine.jar	The Joone's core engine	**Mandatory**
log4j.jar	The configurable logger	**Mandatory**
bsh.jar	The BeanShell interpreter	**Optional**. Needed only if you want to use the scripting features
jakarta-poi.jar	The Jakarta Excel libraries	**Optional**. Needed only if you use the Excel Input/Output synapses
jh.jar	The Java Help libraries	**Never**. Used only in conjunction with the GUI editor contained into the *joone-editor.jar* file
jhotdraw.jar	The drawing framework	**Never**. Used only in conjunction with the GUI editor contained into the *joone-editor.jar* file
visad.jar	The external graphic library to plot graphs	**Never**. Used only in conjunction with the GUI editor contained into the *joone-editor.jar* file

As you can see, only the first two libraries have to be present into your classpath, whereas the next two are needed only if you use some specific feature of the core engine.

The last three libraries, instead, must be used only in conjunction with the GUI editor contained into the *joone-editor.jar* file; but, in this case, you don't need to install manually the editor, as you can use an auto-installer, like depicted in the following paragraph.

2.2.2 The GUI Editor

We have prepared packaged installers for the following platforms[3]:

- Linux[4]
- Windows
- Platform Independent

By using the first two installers, you don't need to be aware about the installation of the Java runtime environment, as they are available both with and without an embedded java virtual machine.
All you need to do is to download the appropriate installer depending on your platform, and run it as described below:

Linux Instructions:
After downloading open a shell and, cd to the directory where you downloaded the installer.
At the prompt type: `sh ./JooneEditorX_Y_Z.bin`
If you do not have a Java virtual machine installed, be sure to download the package which includes one. Otherwise you may need to download one from Sun's Java web site or contact your OS manufacturer.

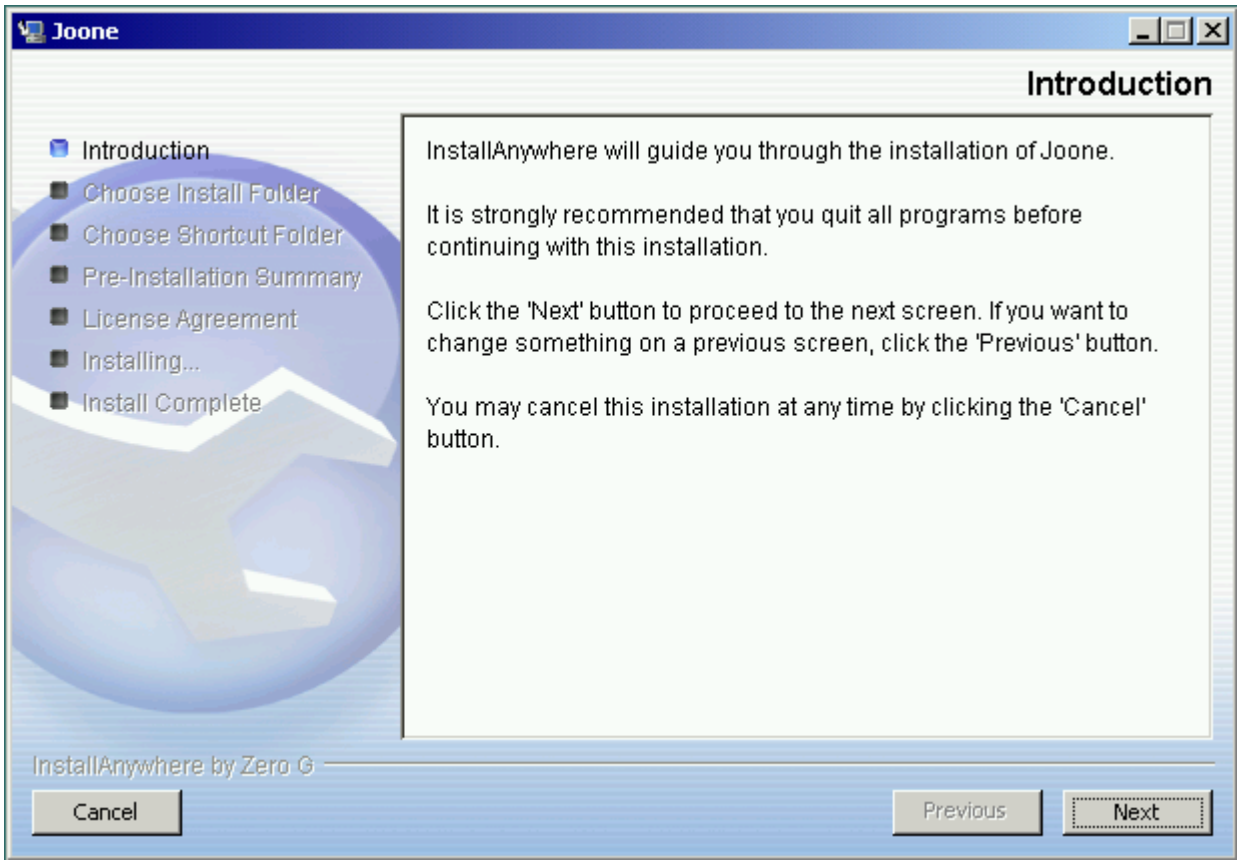
Windows Instructions:
After downloading, double-click JooneEditorX.Y.Z.exe
If you do not have a Java virtual machine installed, be sure to download the package which includes one.

Platform Independent Instructions:
After downloading, expand JooneEditorX.Y.Z-All.zip into a directory (requires a JRE 1.4.2 or later installed), cd to that directory and launch ./RunEditor.sh (Linux/Unix/MacOSX) or RunEditor.bat (Windows).

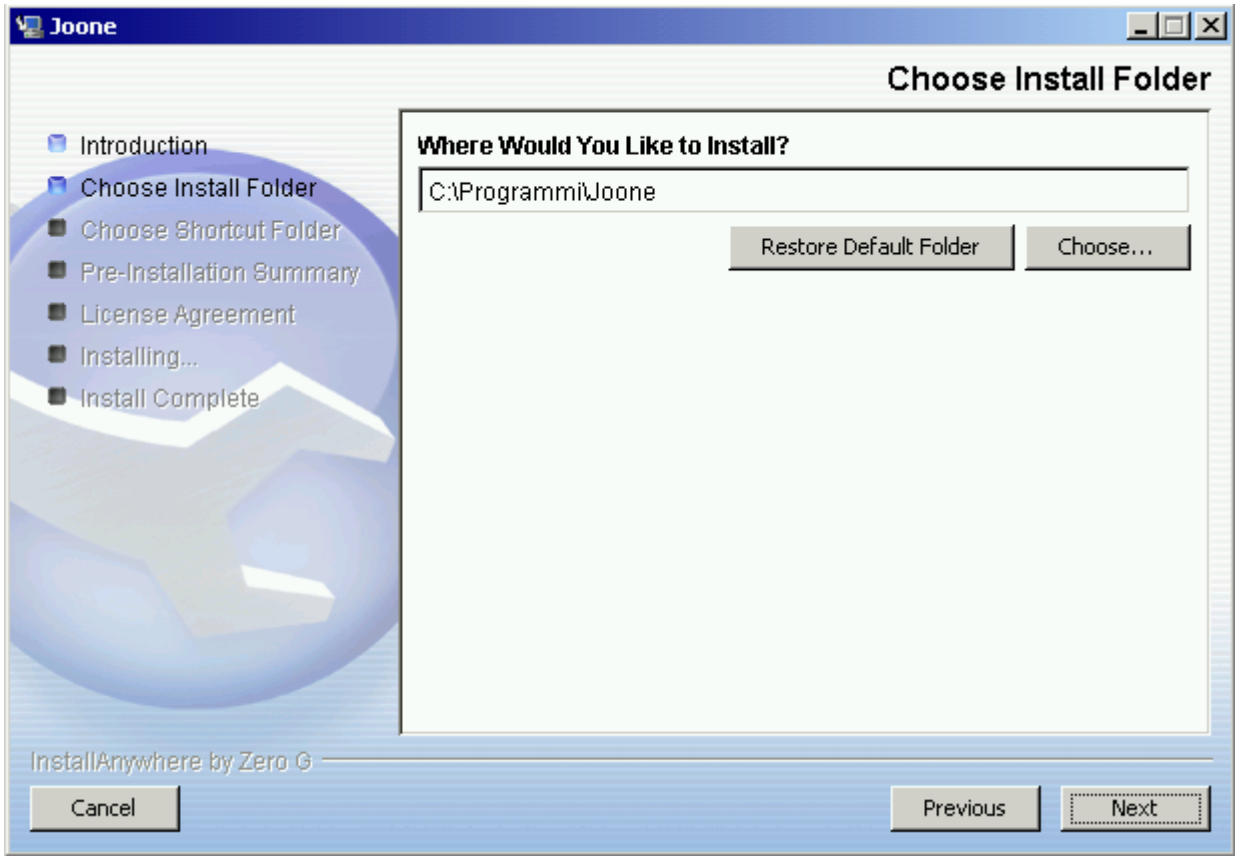
If you have downloaded the Linux or Windows auto-installer, after the launch, you should see the following panel:

[3] InstallAnywhere is a registered trademark of Zero G Software, Inc.
Mac OS is a registered trademark of Apple Computer, Inc.
Solaris and Java are trademarks of Sun Microsystems, Inc.
Windows is a registered trademark of Microsoft Corporation.
All other marks are properties of their respective owners.

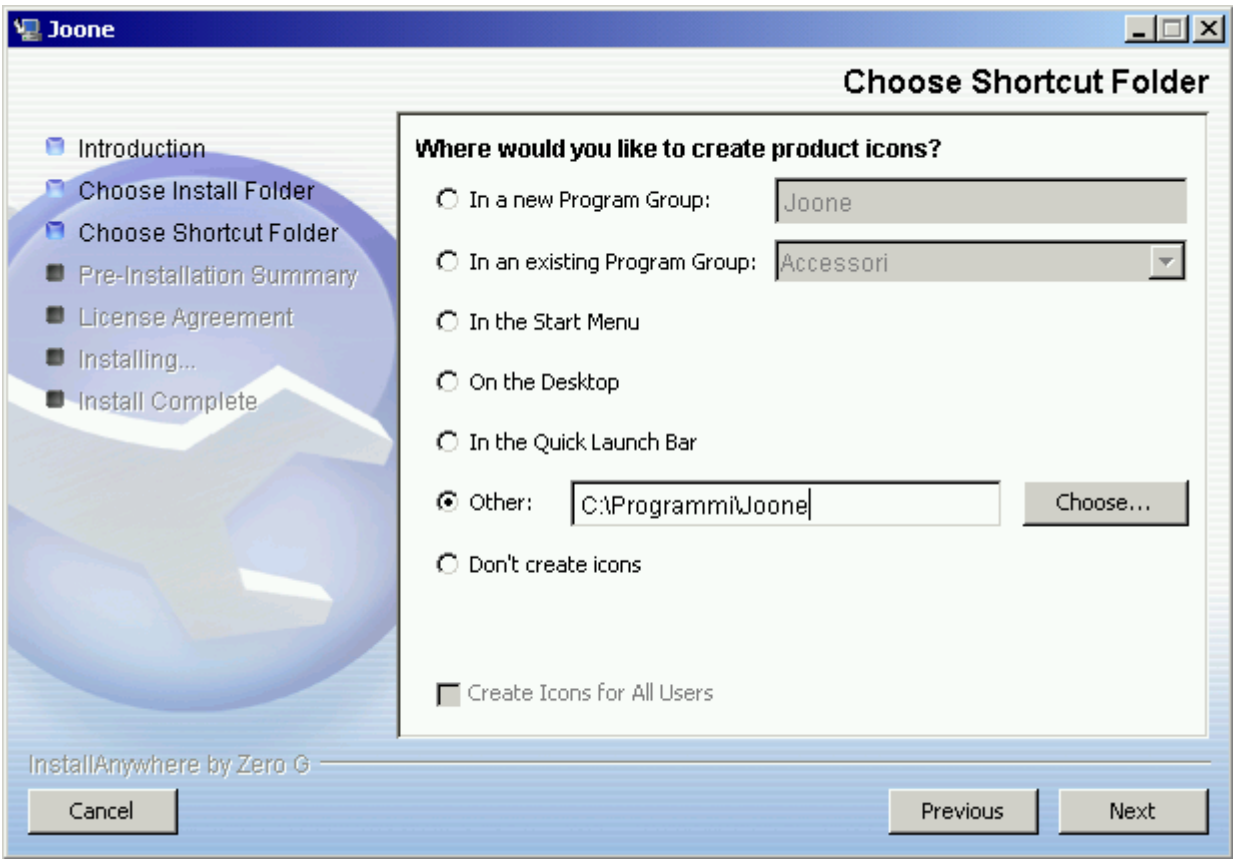
[4] The Linux installer works also on the Solaris OS (of course only that one without the JVM included)



By clicking on the Next button you can advance in the installation process. In any moment, pressing the Cancel button, you can abort and exit from the installation.

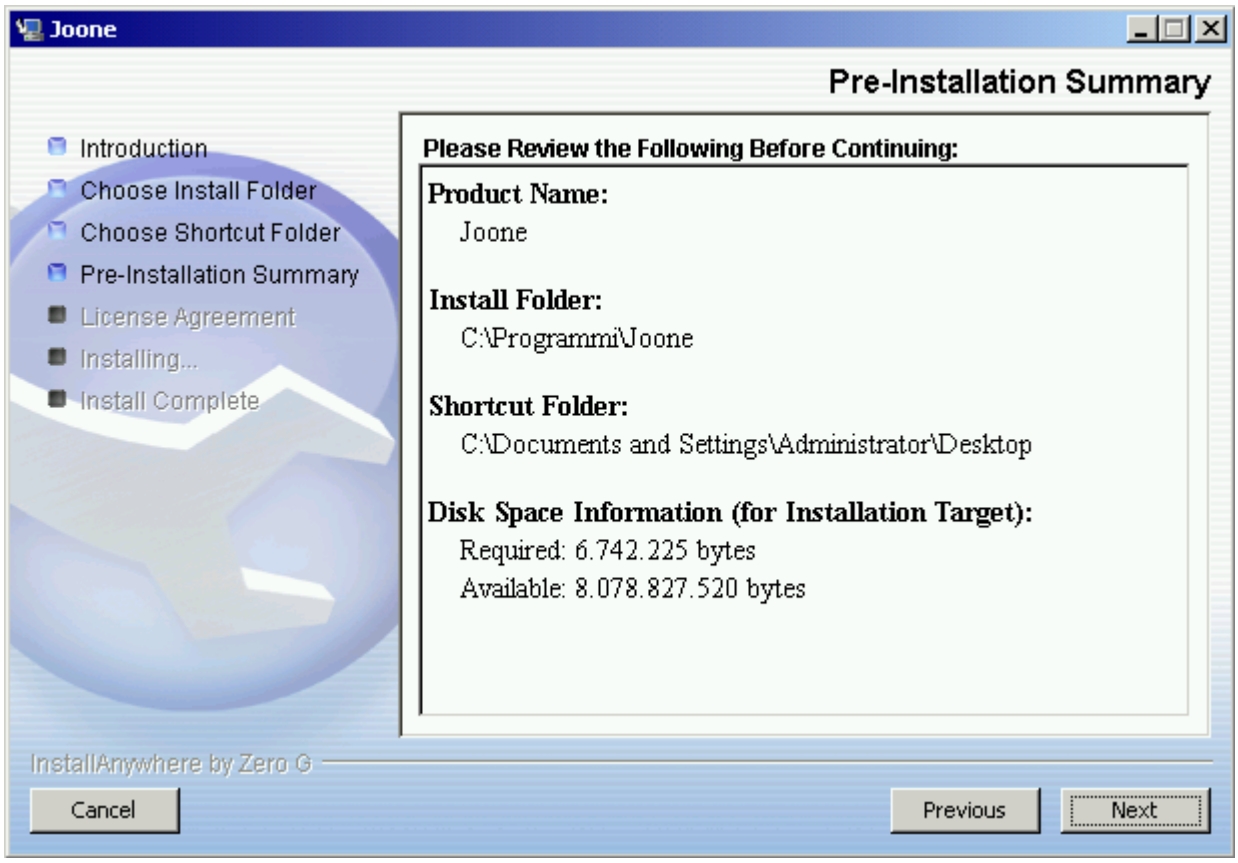


In this panel you must specify the directory where you want to install Joone. The Choose button will open an explorer window, where you can make the choice, whereas using the ‘Restore Default Folder’ button you can reset the directory to its initial value.

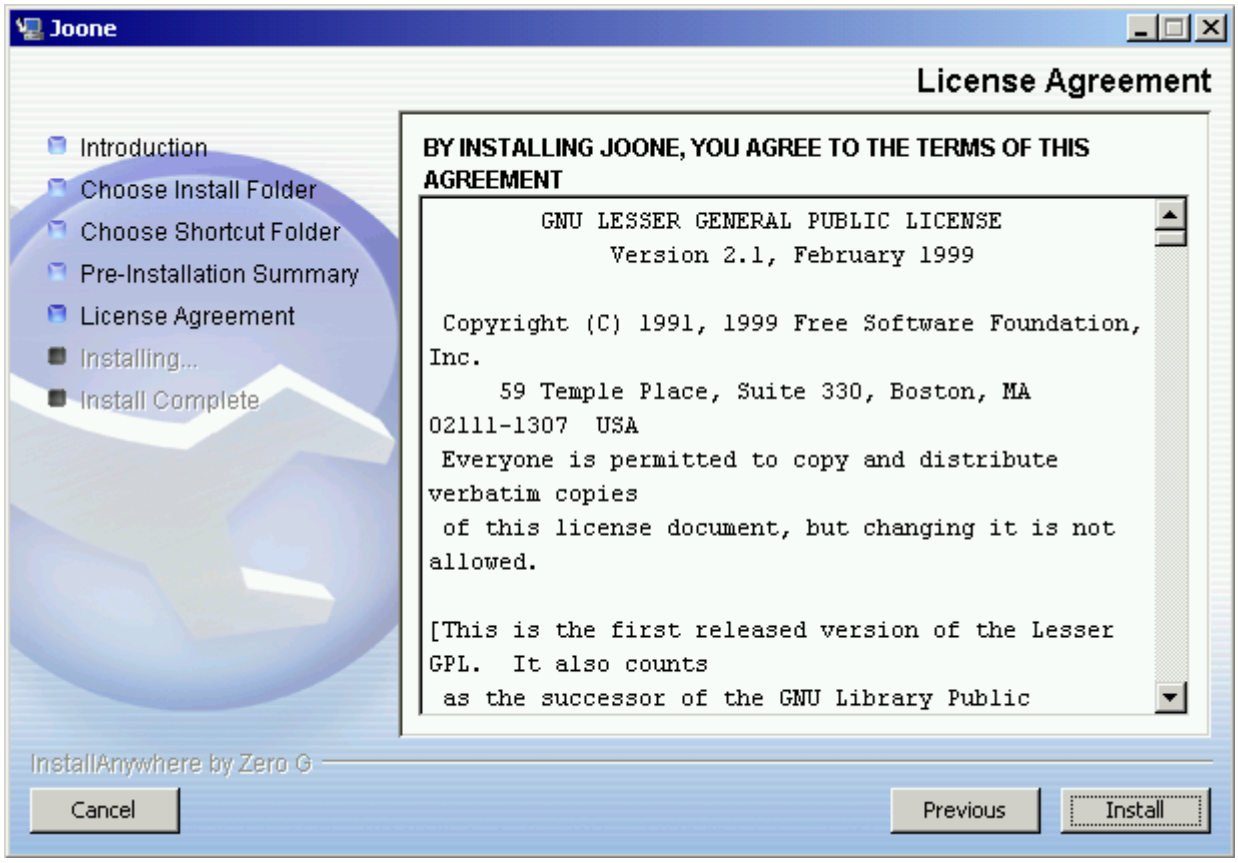


Here you can choose where to put the Joone launcher’s icon.

This panel can contain several available choices depending on the platform where you’re installing on.
By checking the ‘Create Icon for All Users’ box – if not greyed – will give the visibility of the icon to all the users of the system.



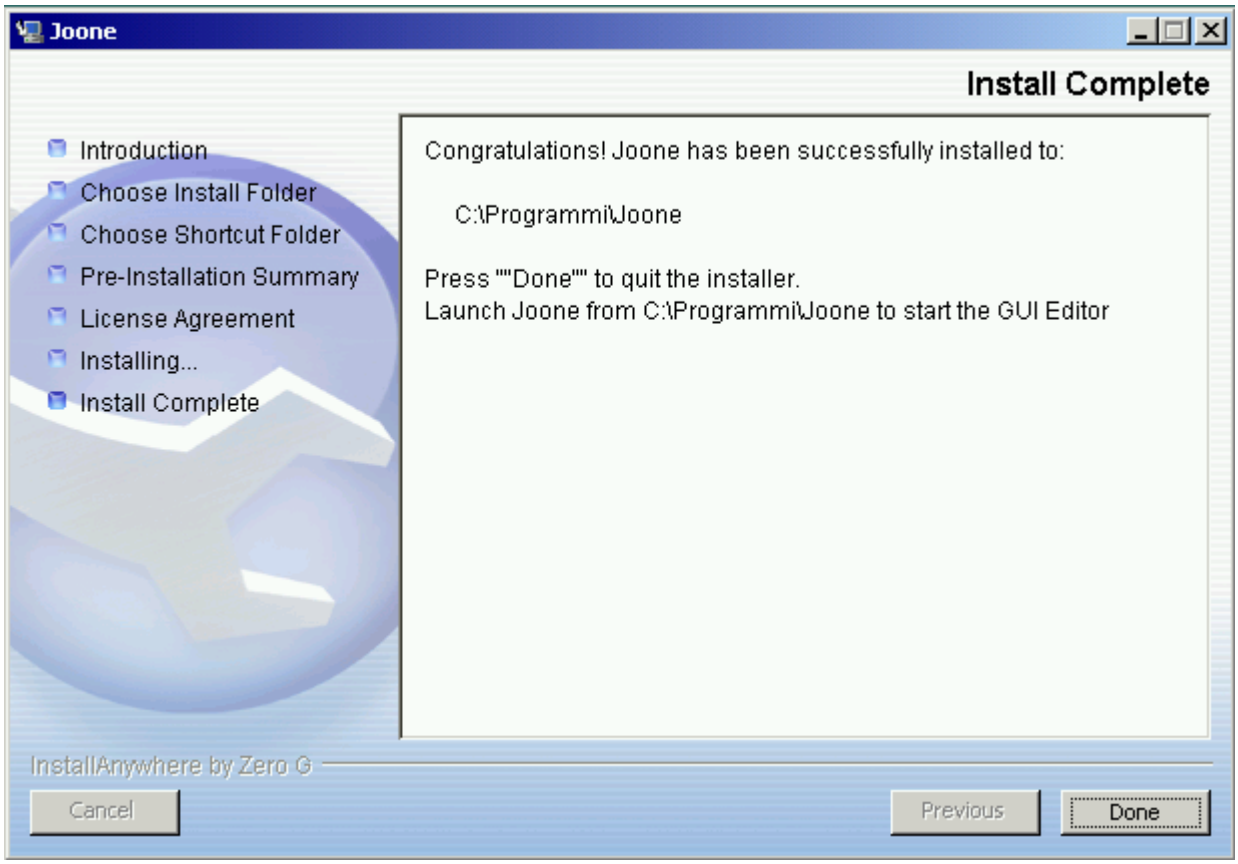
At this point a panel showing the summary of the made choices will appears. If it's all ok, press the Next button, otherwise, pressing the Previous button, you can go back to the previous panels to review and change some parameter.



Now the panel showing the GNU LESSER GENERAL PUBLIC LICENSE, the license under which Joone is released.

Be aware: Open Source **doesn't mean** 'no license', hence, before to continue, you must carefully read the license agreement, and press the 'Install' button only if **you agree to the terms of the license**. A copy of the LGPL license is contained in the last chapter of this paper.

If you continue, the installation process starts and a panel indicating the progress will appear.
At the end, the following panel indicating the success of the operation will be shown.



Press Done to exit.

After the installation, you should found a file named *Joone* (or *Joone.bat* for the Windows platforms) into the chosen installation directory. You must execute it (a double click from within the file explorer should work on all the platforms) to run the editor.
If you have chosen to add a shortcut to the Start Menu or to the Desktop, you can press it to start the application.

2.3 Building from the source distribution

In this paragraph we'll show how to build joone starting from the source distribution, but first of all you need to install on your system some useful tool.

2.3.1 Prerequisites

You need to have installed on your system:

1. a Java Development Kit version 1.4 or above (http://java.sun.com)
2. the ANT build tool v. 1.5.1 or above (http://ant.apache.org)
3. the sources of joone, and to do it, you can either get the last released version, or download the last (unstable) code from the CVS repository.

The instructions to get Java JDK and ANT installed and running on your system go over the scope of this document, but you can read a lot of documentation available on Internet.
Now we'll see how to get the joone's source code.

2.3.2 Getting the last released source code

The released version is preferable if you need to use a stable and tested version of joone, without be worried about possible unknown or not fixed bugs.
To do it, open your preferred browser and simply go to the download page of joone at http://sourceforge.net/project/showfiles.php?group_id=22635 and get the files `joone-engine-x.y.z.zip` (the core engine), `joone-editor-x.y.z.zip` (the GUI editor) and `joone-ext.zip` (the external libraries).
Note: x, y and z are respectively the major/minor version and the build number of the last available distribution.
Unzip them on a directory of your file system (say c:\joone for Windows or ~/joone for Linux).

2.3.3 Getting the CVS sources

If you need to use some new feature of joone still not released, you can get the last developed source code from the CVS repository.
To do it, you need to have a cvs client installed on your system. Unix/Linux systems normally have it already installed, whereas for the Windows system go to http://www.cvshome.org/ and download a suitable version for your OS.
The CVS repository of Joone is hosted at SourceForge, so here is an extract from the instructions gave from SF cvs page:

"...This project's SourceForge.net CVS repository can be checked out through anonymous (pserver) CVS with the following instruction set. The module you wish to check out must be specified as the modulename. When prompted for a password for anonymous, simply press the Enter key.

```
cvs -d:pserver:anonymous@joone.cvs.sourceforge.net:/cvsroot/joone login
cvs -z3 -d:pserver:anonymous@joone.cvs.sourceforge.net:/cvsroot/joone co joone
```

Information about accessing this CVS repository may be found in our document titled, "Basic Introduction to CVS and SourceForge.net (SF.net) Project CVS

Services"
(http://sourceforge.net/docman/display_doc.php?docid=14033&group_id=1).

Updates from within the module's directory do not need the -d parameter.

***NOTE:** UNIX file and directory names are case sensitive. The path to the project CVSROOT must be specified using lowercase characters (i.e. /cvsroot/joone)"*

Anyway you need to download the file containing the external libraries (joone-ext.zip) and unzip it into the same directory where you have checked out from cvs (read at the previous chapter about how to download it).

2.3.4 Compiling

Regardless of which repository you have decided to download from, you should have on your file system the following directory tree:

```
<base_dir>
   <joone>
      <lib>
      <org>
      <joone>
            <data>
            <edit>
            <engine>
            <exception>
            <images>
            <inspection>
            <io>
            <net>
            <samples>
            <script>
            <util>
```

Before to start the build process, you need to edit the build.xml file found in the root installation directory. Open it with a text editor and search the following line:

```
<property name="base" value="/usr/SourceForge"/>
```

change the path into the quotes with your previous chosen installation directory (e.g. c:\\joone or ~/joone) and save the file.
Assuming you have the Java JDK and ANT correctly installed and running (in order to verify, try to launch in a console the commands 'javac' and 'ant'), you need to cd into the installation directory and launch at the prompt the command 'ant'.
At the end of the operation, under the installation directory, if no error occurs, you should have a subdirectory named 'build' containing all the compiled classes.
At this point, to run the GUI editor, you need to:

1. Put the <base_dir>/build directory and all the <base_dir>/lib/*.jar files on your classpath
2. Open a console and launch the following command: java org.joone.edit.JoonEdit

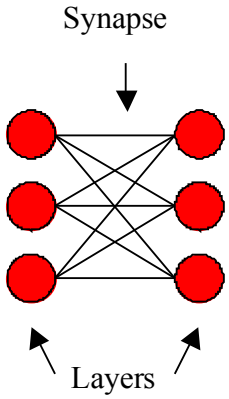
The main window of the editor should appear.

3 Inside the Core Engine

3.1 Basic Concepts

Each neural network (NN) is composed of a number of components (**layers**) connected together by connections (**synapses**). Depending on how these components are connected, several neural network architectures can be created (feed forward NN, recurrent NN, etc).
This section deals with feed forward neural networks (FFNN) for simplicity's sake, but it is possible to build whatever neural network architecture is required with Joone.

A FFNN is composed of a number of consecutive layers, each one connected to the next by a synapse. In a FFNN recurrent connections from a layer to a previous one are not permitted. Consider the following figure:



This is a sample FFNN with two layers fully connected with synapses. Each layer is composed of a certain number of neurons, each of which have the same characteristics (transfer function, learning rate, etc).

A neural net built with Joone can be composed of whatever number of layers belonging to different typologies (linear, sigmoid, ect.).
Each layer processes its input signal by applying a transfer function and sending the resulting pattern to the synapses that connect it to the next layer. So a neural network can process an input pattern, transferring it from its input layer to the output layer.

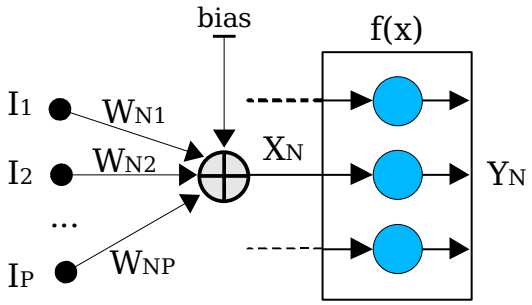
This is the basic concept upon which the entire engine is based.

3.2 The Transport Mechanism

Note: as of Joone v. 2.0, a new single-thread engine has been written, in order to improve the performances in presence of machines with multi-core CPUs. As a conseguence, the Layer doesn't run anymore within its own separate thread, so all the concepts described below, even if still true when the network is launched in multi-thread mode, don't apply completely when the new single-thread engine is used.

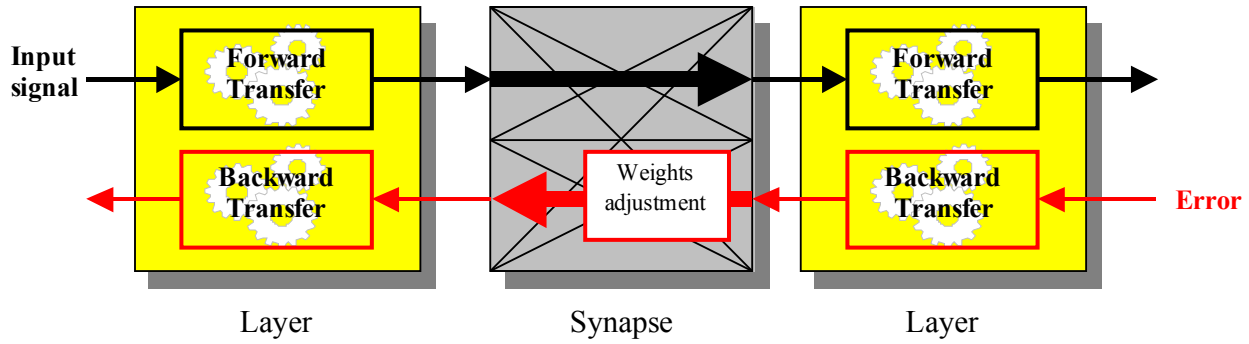
In order to ensure that it is possible to build whatever neural network architecture is required with Joone, a method to transfer the patterns through the net is required without the need of a central point of control.

To accomplish this goal, each layer of Joone is implemented as a *Runnable* object, so each layer runs independently from the other layers (getting the input pattern, applying the transfer function to it and putting the resulting pattern on the output synapses so that the next layers can receive it, processing it and so on) as depicted by the following basic illustration:



Where for each neuron N:
X_N – The weighted net input of each neuron = (I1 * W_N1) + … + (I_P * W_{NP}) + bias
Y_N – The output value of each neuron = f(X_N)
f(X) – The transfer function (depending on the kind of layer used)

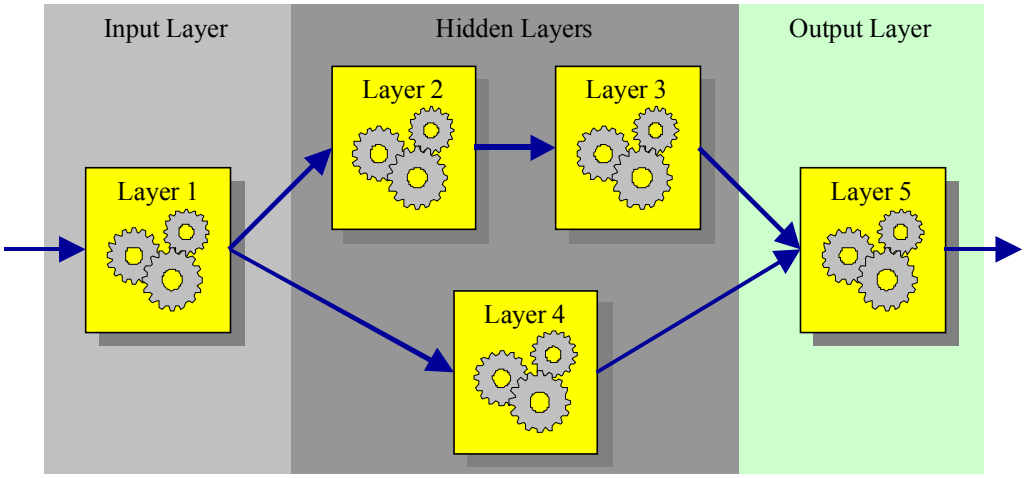
This transport mechanism is also used to bring the error from the output layers to the input layers during the training phases, allowing the weights and biases to be changed according to the chosen learning algorithm (for example the backprop algorithm).
In other words, the Layer object alternately ‘pumps’ the input signal from the input synapses to the output synapses, and the error pattern from the output synapses to the input synapses.
To accomplish this, each layer has two opposing transport mechanisms, one from the input to the output to transfer the input pattern during the recall phase, and another from the output to the input to transfer the learning error during the training phase, as depicted in the following figure:



Each Joone component (both layers and synapses) has its own pre-built mechanisms to adjust the weights and biases according to the chosen learning algorithm.
Complex neural network architectures can be easily built, either linear or recursive, because there is no necessity for a global controller of the net.

Imagine each layer acts as a pump that 'pushes' the signal (the pattern) from its input to its output, where one or more synapses connect it to the next layers, regardless of the number, the sequence or the nature of the layers connected.
This is the main characteristic of Joone, guaranteed by the fact that each layer runs on its own thread, representing the unique active element of a neural network based on the Joone's core engine.

Look at the following figure (the arrows represent the synapses):



In this manner any kind of neural networks architecture can be built.

To build a neural network, simply connect each layer to another as required using a synapse, and the net will run without problems. Each layer (running in its own thread) will read its input, apply the transfer function, and write the result to its output synapses, to which there are other layers connected and running on separate threads, and so on.

Joone allows any kind of net to be built thanks to its modular architecture, like a LEGO® bricks system

---

By this means:

- **The engine is flexible**: you can build any architecture you want simply by connecting each layer to another with a synapse, without being concerned about the architecture. Each layer will run independently, processing the signal on its input and writing the results to its output, where the connected synapses will transfer the signal to the next layers, and so on.
- **The engine is scalable**: if you need more computation power, simply add more CPU to the system. Each layer, running on a separated thread, will be processed by a different CPU, enhancing the speed of the computation.
- **The engine closely mirrors reality**: conceptually, the net is not far from a real system (the brain), where each neuron works independently from each other without a global control system.

All the above characteristics are valid also for the single-thread engine (introduced with the version 2.0 of Joone), where the Layers don't run within their own separate threads, but instead are invoked from an unique external thread, that is instantiated and handled by the NeuralNet object. The reengineering of the core engine has been studied in order to expose the same features of the multi-thread version, maintaining so an almost complete compatibility with the past releases.

3.3 The Processing Elements

Now we'll see the principal kind of layers and synapses implemented into the core engine, and for everyone we'll show the transfer function and the most common usage.

3.3.1 The Layers

The Layer object is the basic element that forms the neural net.
It is composed of neurons, all having the same characteristics. This component transfers the input pattern to the output pattern by executing a transfer function.

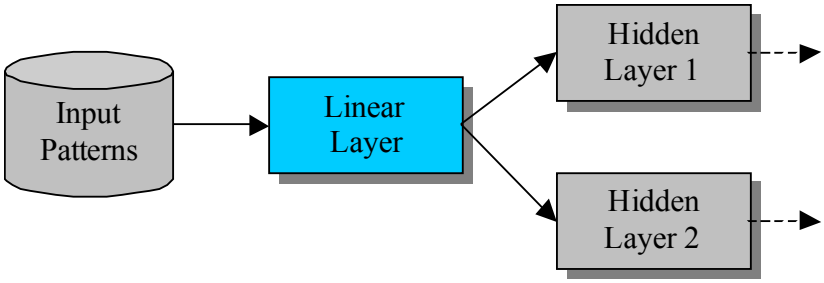
The output pattern is sent to a vector of Synapse objects attached to the layer's output. It is the active element of a neural network in Joone, in fact it runs in a separated thread (it implements the *java.lang.Runnable* interface) so that it can run independently from other layers in the neural net.

3.3.1.1 The Linear Layer

Description

The Linear Layer is the simplest kind of layer, as it simply transfers the input pattern to the output applying a linear transformation, i.e. multiplying it by a constant term, the Beta term. If it is equal to 1 (one), then the input pattern is transferred without modifications.

The Linear Layer is commonly used as a buffer, placed, for instance, as the first layer of a neural network to permit to send an unmodified copy of the input patterns to several hidden layers, as depicted in the following figure:



Without a Linear Layer, in these cases it would be impossible to send the same input pattern to many subsequent layers, because the input component (the InputSynapse here represented by a cylinder) can be attached only to one layer.

Transfer Function

$$y = \beta \cdot x$$

3.3.1.2 The Biased Linear Layer

Description

The Biased Linear Layer is an extension of the Linear Layer, because it applies a linear transfer function to its input, but differs from Linear Layer in two ways:

- The Biased Linear Layer, as its name says, uses biases. The biases can/will be adjusted during the training phase
- It has no scalar beta parameter

This layer can be used wherever you need a layer having a linear transfer function, but also having an adjustable parameter – the bias – that adapts the response of the layer according to the gradient, that is back-propagated during the learning process.

Transfer Function

$$y = x + bias_n$$

where $bias_n$ is the bias of the n_{th} neuron.

3.3.1.3 The Sigmoid Layer

Description

The Sigmoid Layer applies a sigmoid transfer function to its input patterns, representing a good non-linear element to build the hidden layers of the neural network.
The sigmoid layer can be used to build whatever layer of a neural network.
Its output is smoothly limited within the range 0 and 1.

Transfer Function

$$y = \frac{1}{1 + e^{-x}}$$

3.3.1.4 The Tanh Layer

Description

The Tanh Layer is similar to the Sigmoid Layer except that the applied function is a hyperbolic tangent function, that limits its output within the range –1 and +1.

Transfer Function

$$y = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

3.3.1.5 The SoftMax Layer

This layer is similar to the SigmoidLayer, as the output of each node ranges from 0 and 1, but the difference is that the sum of all the nodes is always 1.
Due to this characteristic, the output values of the SoftMax layer can be interpreted as a posterior probability, i.e. the activation of each output node represents the probability the input pattern belongs to the corresponding output class (represented by each output node).
This class is used in supervised networks to implement the 1 of C classification (by contrast with the SigmoidLayer, that is normally used for binary classification; the statisticians usually call softmax a “multiple logistic” function, as it reduces to logistic function when there are only two output categories).

Transfer Function

$$y = \frac{e^{x}}{\sum_{j=1}^{C} e^{x_j}}$$

3.3.1.6 The Logarithmic Layer

Description

This layer applies a logarithmic transfer function to its input patterns, resulting in an output that, unlike from the above two previous layers, ranges from 0 to $+\infty$

This behaviour permits to avoid the saturation of the processing elements of a layer in presence of a lot of input synapses connected, or in presence of input values very near to the limits 0 and 1, where the sigmoid and tanh layers have a response curve very flat.

Transfer Function
y = log(1 + x) if $x \geqslant 0$
y = log(1 – x) if $x < 0$

3.3.1.7 The Sine Layer

Description

The output of a Sine Layer neuron is the sum of the weighted input values, applied to a sine – sin(x) – transfer function. Neurons with sine activation function might be useful in problems with periodicity.

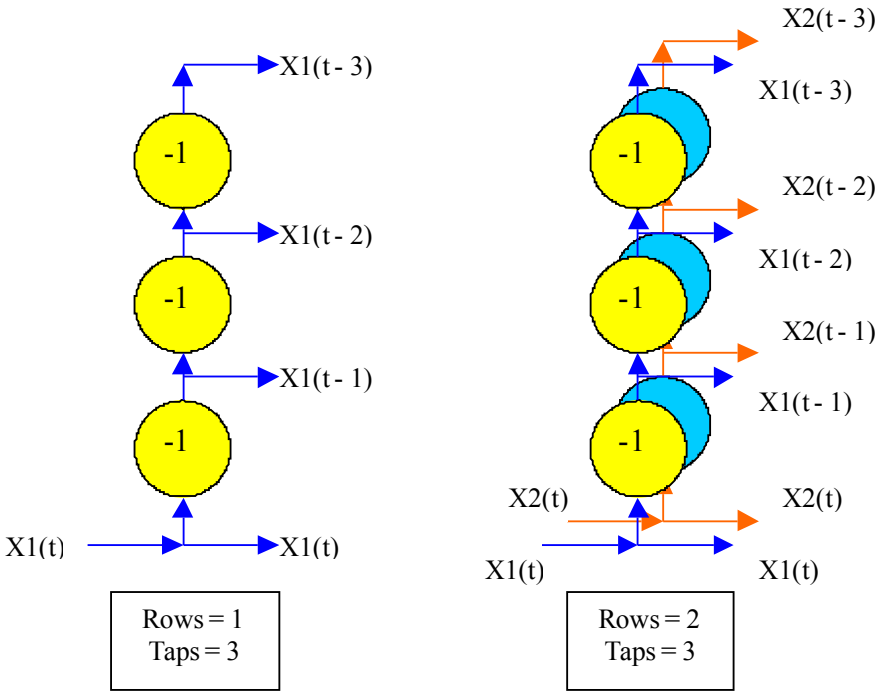
Transfer Function
y = sin(x)

3.3.1.8 The Delay Layer

Description

The delay layer applies the sum of the input values to a delay line, so that the output of each neuron is delayed a number of iterations specified by the taps parameter.

To understand the meaning of the taps parameter, look at the following picture that contains two different delay layers, one with 1 rows and 3 taps, and another with 2 rows and 3 taps:



the delay layer has:

- the number of inputs equal to the rows parameter
- the number of outputs equal to rows * (taps + 1)

The taps parameter indicates the number of output delayed cycles for each row of neurons, plus one because the delayed layer also presents the actual input signal Xn(t) to the output. During the training phase, error values are fed backwards through the delay layer as required.

This layer is very useful to train a neural network to predict a time-series, giving it a 'temporal window' of the input raw data.

Transfer Function

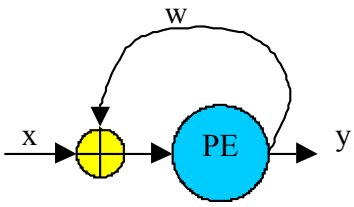
$$y_N = x_{(t-N)}$$

where: $0 < N \leqslant taps$

3.3.1.9 The Context Layer

Description

The context layer is similar to the linear layer except that it has an auto-recurrent connection between its output and input, like depicted in the following figure:



The recurrent weight *w* is named 'timeConstant' because it back-propagates the past output signals and, as its value is less than one, the contribute of the past signals decays slowly toward zero at each cycle. Its value is constant, hence doesn't change during the training phase.
In this manner the context layer has a own 'memory' embedded mechanism.
This layer is used in recurrent neural networks like the Jordan-Elman ones.

Transfer Function

$$y = \beta \cdot (x + y_{(t-1)} \cdot w)$$

where:

β = the beta parameter (inherited from the linear layer)
w = the fixed weight of the recurrent connection (not learned)

3.3.1.10 The GaussLayer

Description
The output of a GaussLayer neuron is the sum of the weighted input values, applied to a gaussian function. It is useful whenever the layer must respond to a particular set of input patterns (i.e. which ones having their coords within the center of the gaussian curve). This layer has a curve centered on zero, and its size is not adjustable. If you're searching for a gaussian component to use in a RBF or Kohonen network, see the note below.

Transfer Function

$$y = e^{(-x*x)}$$

Note: do not confuse this layer neither with the GaussianLayer, nor with the RBFGaussianLayer. The first one is used as output map of a Kohonen SOM, whereas the RBF one must be used as hidden layer of a RBF network.

3.3.1.11 The RBFGaussianLayer

This class implements the nonlinear layer in Radial Basis Function (RBF) networks using Gaussian functions.
Its output, like the GaussLayer described above, is the sum of the weighted input values, applied to a gaussian function. The difference is represented by the gaussian curve itself, that has adjustable its own center, according to predefined parameters.
The center of the gaussian curve can be adjusted according two different techniques:

1. **Random** – each node will be assigned a randomly chosen center according the input data. In order to obtain that, a new plugin – the RbfRandomCenterSelector – has been built. This plugin must be attached to the input synapse of the network, and it will calculate the mean (center) and the std deviation for each node of the layer.
2. **Custom** – each node will be assigned a predefined center, and to do this a new component – the RbfGaussianParameters – has been built. The calling application must set the mean (center) and the std deviation for each node of the layer before to start the training phase.

Transfer Function

$$y = e^{\frac{D^2}{(-\sigma(x)*\sigma(x))}}$$

Where D^2 is the squared euclidean distance and σ(x) is the std deviation calculated on the input patterns.

Note: the XOR_static_RBF.java class in the org.joone.samples.engine.xor.rbf package contains a complete example that shows how to build and use a RBF network.

The actual implementation of the RBF network is not complete, as it represents only a starting point (a good starting point, I think, thanks to Boris Jansen), therefore anyone interested to complete the work is welcome.

3.3.1.12 The WinnerTakeAll Layer

Description

The WinnerTakeAll layer is one of the components – along with the GaussianLayer and the KohonenSynapse – useful to build unsupervised self-organized-map (SOM) networks.
This kind of networks learns without an external teacher, simply by detecting the similarities of the input patterns and categorizing (i.e. projecting) them on a dimensionally reduced (1D or 2D) map.

This layer implements the Winner Takes All SOM strategy. The layer expects to receive Euclidean distances between the previous synapse (the KohonenSynapse) weights and it's input. The layer simply works out which node is the winner and passes 1.0 for that node and 0.0 for the others.

In this manner the attached KohonenSynapse can adjust its own weights according to the winner neuron, updating the internal connections so that, when a similar input is presented, the same neuron will be activated (or one near it, depending on how much that pattern is similar to that seen during the learning phase).

Transfer Function

$y_n = 1$ if *n* is the most active neuron,
$y_n = 0$ otherwise

3.3.1.13 The Gaussian Layer

Description

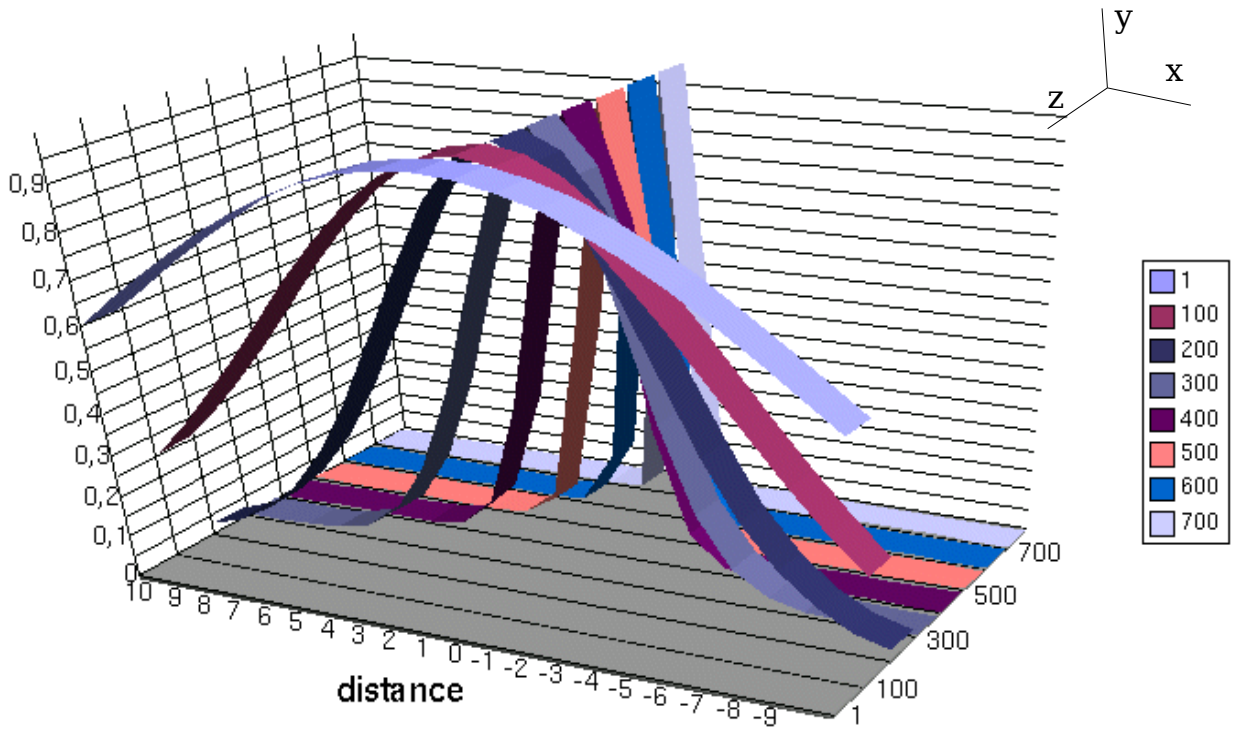
The Gaussian layer performs a similar work like the WTA layer, but in this case it activates the output neurons according a gaussian shape centered around the most active neuron (the winner).

This layer implements the Gaussian Neighborhood SOM strategy. It receives the Euclidean distances between the input vector and weights and calculates the distance fall off between the winning node and all other nodes. These are passed back allowing the previous synapse (the KohonenSynapse) to adjust it's weights.

The distance fall off is calculated according to a Gaussian distribution from the winning node.
In this manner the in the KohonenSynapse not only the weights feeding the winner neuron will be adjusted, but also its neighbor, with a strength inversely proportional to the distance from the winner neuron.

Transfer Function

Better than by a complex formula, the transfer function can be represented by a graphic representation of the output values in correspondence of both the distance from the winner node and the actual epoch.
The neighborhood around the winner node starts very large and then is reduced following a gaussian curve, as depicted in the following image:



the curves represent how the neighborhood function changes during the training epochs; the X axis represents the distance from the winner node (± 10 in this example), the Y axis contains the output values of the layer, and the numbers in the legend (put on the Z axis) represent the number of epochs (from 1 to 700 in this example).
As you can see, an initial phase exists, within which the algorithm maintains the neighborhood size large in order to permit a large number of weights to participate to the adjustments (this phase is named *ordering phase*), after which the neighborhood is maintained very small (the weights are frozen after they have chosen the input vectors to which to respond).
A similar mechanism is implemented into the KohonenSynapse object.

3.3.2 The Synapses

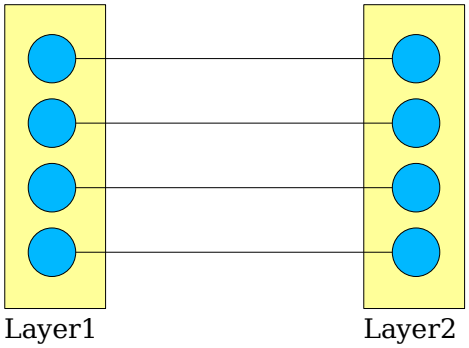
The Synapse represents the connection between two layers, permitting a pattern to be passed from one layer to another.

The Synapse is also the 'memory' of a neural network. During the training process the weigh of each connection is modified according the implemented learning algorithm.
Remember that, as described above, a synapse is both the output synapse of a layer and the input synapse of the next connected layer in the NN, hence it represents a shared resource between two Layers (no more than two, because a Synapse can be attached only once as input or output of a Layer).

To avoid a layer trying to read the pattern from its input synapse before the other layer has written it, the shared synapse in synchronized; in other terms, a semaphore based mechanism prevents two Layers from accessing simultaneously to a shared Synapse.

3.3.2.1 The Direct Synapse

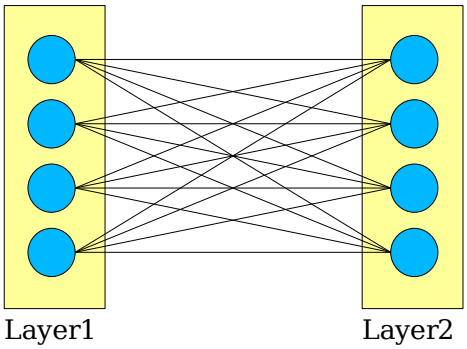
The DirectSynapse represents a direct connection 1-to-1 between the nodes of the two connected layers, as depicted in the following figure:



Each connection has a weight equal to 1, and it doesn't change during the learning phase.
Of course, a DirectSynapse can connect only layers having the same numbers of neurons, or nodes.

3.3.2.2 The Full Synapse

The FullSynapse connects all the nodes of a layer with all the nodes of the other layer, as depicted in the following figure:



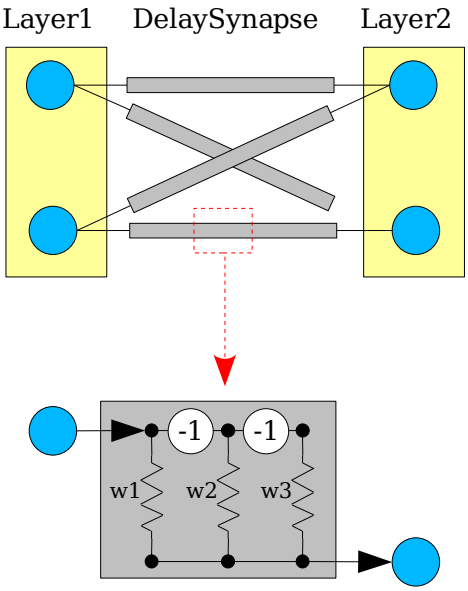
This is the most common type of synapse used in a neural network, and its weights change during the learning phase according to the implemented learning algorithm.

It can connect layers having a whatever number of neurons, and the number of the weights contained is equal to N_1 x N_2, where N_x is the number of nodes of the $Layer_x$

3.3.2.3 The Delayed Synapse

This Synapse has an architecture similar to which of the FullSynapse, but each connection is implemented using a matrix of FIR Filter elements of size NxM.

The following figure illustrates how a DelaySynapses can be represented:

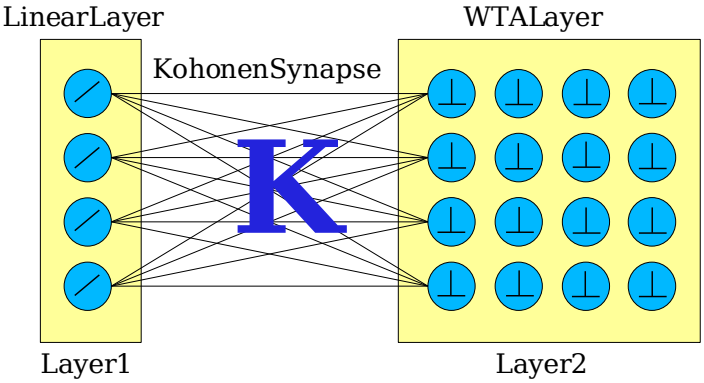


FIR Filter

As you can see in the first figure, each connection – represented with a greyed rectangle - is implemented as a FIR (Finite Impulse Response) filter and in the second figure the internal detail of a FIR filter is shown.
A FIRFilter connection is a delayed connection that permits to implement a temporal backprop algorithm functionally equivalent to the TDNN (Time Delay Neural Network), but in a more efficient and elegant manner.
To learn more on this kind of synapses, read the article *Time Series Prediction Using a Neural Network with Embedded Tapped Delay-Lines*, Eric Wan, in Time Series Prediction: Forecasting the Future and Understanding the Past, editors A. Weigend and N. Gershenfeld, Addison-Wesley, 1994. Moreover, at http://www.cs.hmc.edu/courses/1999/fall/cs152/firnet/firnet.html you can find some good examples using FIR filters.

3.3.2.4 The Kohonen Synapse

The KohonenSynapse belongs to a special kind of components that permit to build unsupervised neural networks.
This components, in particular, is the central element of the SOM (Self Organizing Maps) networks. A KohonenSynapse must be followed necessarily by a WTALayer or a GaussianLayer component, forming so a complete SOM, like depicted in this figure:



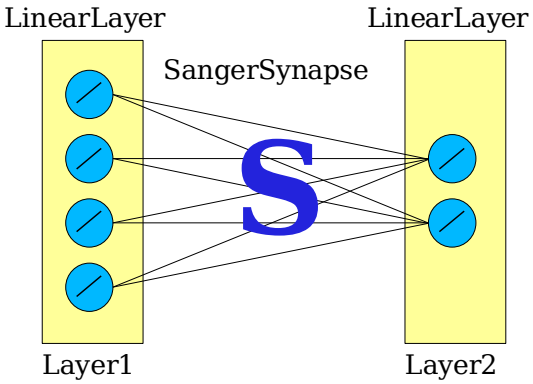
As you can see, a SOM is composed normally by three elements:
1. A LinearLayer that is used as input layer
2. A WTALayer (or GaussianLayer) that's used as output layer
3. A KohonenSynapse that connects the two above layers

During the training phase, the KohonenSynapse's weights are adjusted to map the N-dimensional input patterns to the 2D map represented by the output synapse.

What is the difference between the WTA and the Gaussian layers? The answer is very simple, and depends on the precision of the response we want from the network.
If we're, for instance, using a SOM to make predictions (for instance to forecast the next day's weather), probably we need to use a GaussianLayer as output, because we want a response in terms of percentage around a given value (it will be cloudy and maybe it will rain), whereas if we're using a SOM to recognize handwritten characters, we need a precise response, (i.e. '*the character is A*', NOT '*the character could be A or B*') hence in this case we need to use a WTALayer, that activates one (and only one) neuron for each input pattern.

3.3.2.5 The Sanger Synapse

The SangerSynapse serves to build unsupervised neural networks that apply the PCA (Principal Component Analysis) algorithm.
The PCA is a well known and widely used technique that permits to extract the most important components from a signal. The Sanger algorithm, in particular, extracts the components in ordered mode – from the most meaningful to the less one – so permitting to separate the noise from the true signal.
This components, by reducing the number of input values without diminishing the useful signal, permits to train the network on a given problem reducing considerably the training time.
The SangerSynapse normally is posed between two LinearLayers, and the output layer has less neurons than the input layer, as depicted in the following figure:



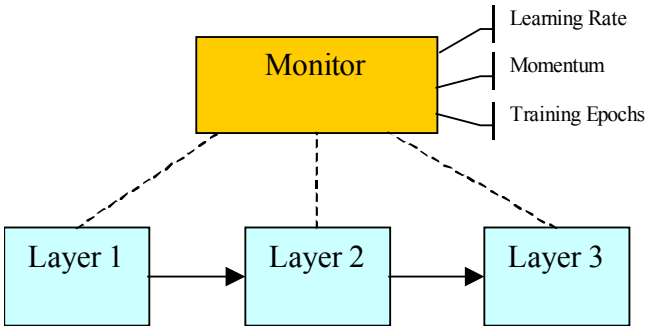
By using this synapse along with the Nested Neural Network component it's very easy to build modular neural networks where the first NN acts as a pre-processing element that reduces the number of the input columns and consequently its noise.

3.4 The Monitor: a central point to control the neural network

Obviously a neural network can't be composed only by the above two kinds of processing elements - layers and synapses - because there is the necessity to control all the parameters interested in the running and/or training process.
For this purpose the engine is composed by several other components designed to provide the neural network with a series of services.

The main component that is ever present in each joone's based neural network is the **Monitor** object. It represents the central point within which are contained all the parameters needed by the other components to work properly, like the learning rate, the momentum, the number of training epochs, the current cycle, etc.

Each component of a neural network (both layers and synapses) receive a pointer to an instance of the monitor object. This instance can be different for each component, but usually only a unique instance is created and used, so that each component can access to the same parameters for the entire neural network, as depicted in the following figure:



In this manner, when the user wants to change any of such parameters, s/he must simply change the corresponding value in the Monitor object; as a result

each component of the neural network will receive the new value of the changed parameter.

The Monitor provides services not only to the internal components of a neural network, but also to the external application that uses it.
The Monitor object, in fact, provides any external application with a notification mechanism based on several events raised when a particular action is performed. For instance, an external application can be advised when the neural network starts or stop the training epochs, when it finishes a cycle or when the value of the global error (the RMSE) changes during the training phase.

In this manner any application using Joone can asynchronously perform a certain action in response of a specific event of the controlled neural network as, for instance, to stop the training when the desired RMSE is reached, or to check the generalisation level of the net using a separate input validation set, or to display in some graphical window the actual values of the parameters of the net, …and so on

The following is a list of the Monitor object's features.

3.4.1 The Monitor as a container of the Network Parameters

The Monitor contains all the parameters needed during the training phases, e.g. the learning rate, the momentum, etc. Each parameter has its own getter and setter method, conforming to the JavaBeans specifications.
These parameters can be used by an external application, for example, to display them in a user interface, or by an internal component to calculate the formulas to implement the recall/training phases, representing in this way a standard and centralized mechanism for getting and setting the parameters needed for its work.

3.4.2 The Monitor as the Network Controller

The Monitor object is also a central point for controlling the start/stop times of a neural network.
It has some parameters that are useful to control the behaviour of the NN, e.g. the total number of epochs, the total number of the input patterns, etc.
Before explaining how does this works, an explanation is required of how the input components of a neural network work.
When the first Layer of a neural network calls its connected InputSynapse component to read a pattern from an external source (see the I/O components chapter), this object calls the Monitor to advise it that a new cycle must be processed.
The Monitor, according to its internal state (current cycle, current epoch, etc.), verifies if the next input pattern must be normally processed.

If yes, the InputSynapse simply receives the permission to continue to elaborate the next pattern, and all the counters internal to the Monitor object are updated.
If no (i.e. the net reached the last epoch), the Monitor object doesn't give the permission to continue and also it notifies all the external applications raising an event that describes the nature of the notification.

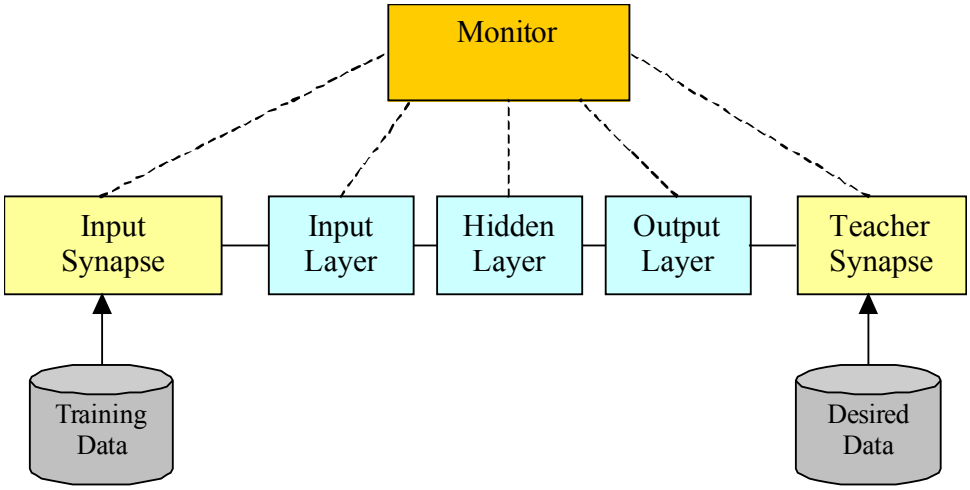
In this manner the following services are made available using the Monitor object:

1. The `InputSynapse` knows if it can read and process the next input pattern (otherwise it stops), being advised by the returned Boolean value.
2. An external application (or the NeuralNet object itself) can start/stop a neural network simply by setting the initial parameters of the Monitor. To simplify these actions, some simple methods - Go (to start), Stop (to stop) and runAgain[5] (to restore a previous stopped network to running) - have been added to the Monitor.
3. The observer objects (e.g. the main application) connected to the `Monitor` can be advised when a particular event raises, as when an epoch or the entire training process has finished (for example either to show to the user the actual epoch number or the actual training error).
   To see how to manage the events of the `Monitor` to read the parameters of the neural network, read the following paragraph.

[5] As far Joone 2.0, the listed Monitor's methods are substituted by the NeuralNet go(), stop() and restore() methods respectively. See the Chapter 7 for more details.

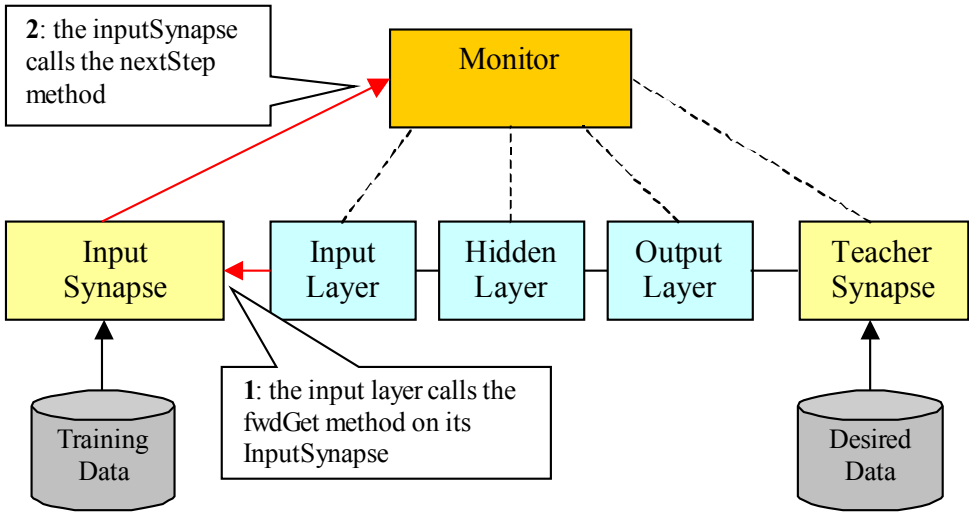
3.4.3 Managing the events

In order to explain how the events of the Monitor object can be used by an external application, this paragraph explains in detail what happens when a neural network is trained and when the last epoch is reached.

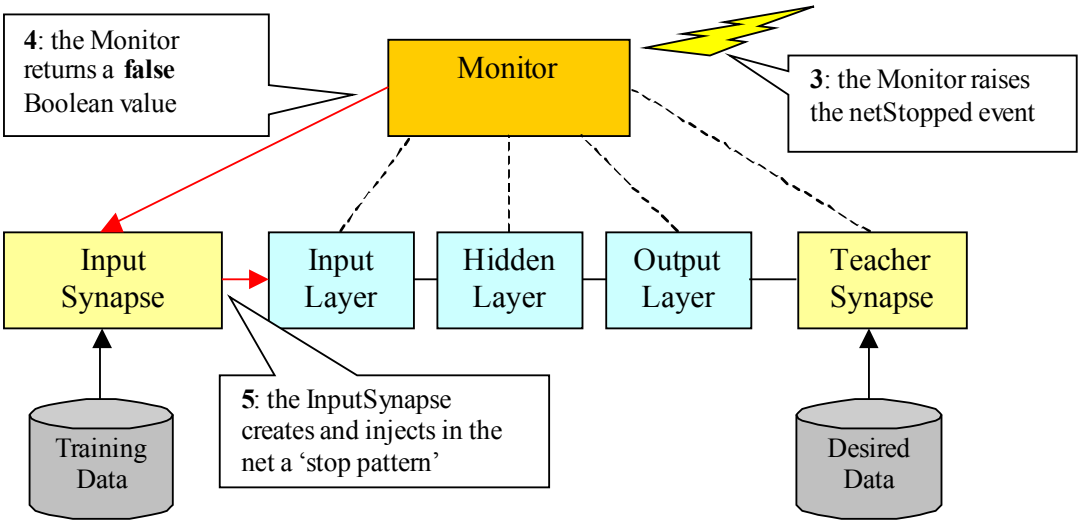


Suppose to have a neural network composed, as depicted in the above figure, by three layers, and an InputSynapse to read the training data, a TeacherSynapse to calculate the error for the backprop algorithm, and a Monitor object that controls the overall training process. As already mentioned, all the components of a neural network built with Joone obtain a reference to the Monitor object, represented in the figure by the dotted lines.

Supposing the net is started in training mode, in the following figures all the phases involved in the process are shown when the end of the last epoch is reached. The numbers in the label boxes indicate the sequence of the processing:

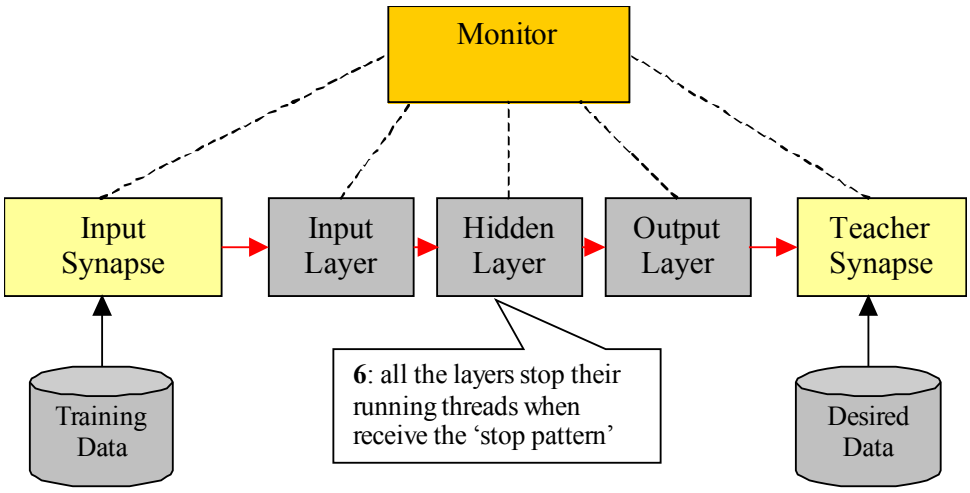


When the input layer calls the InputSynapse (1), the called object interrogates the Monitor to know if the next pattern must be processed (2).



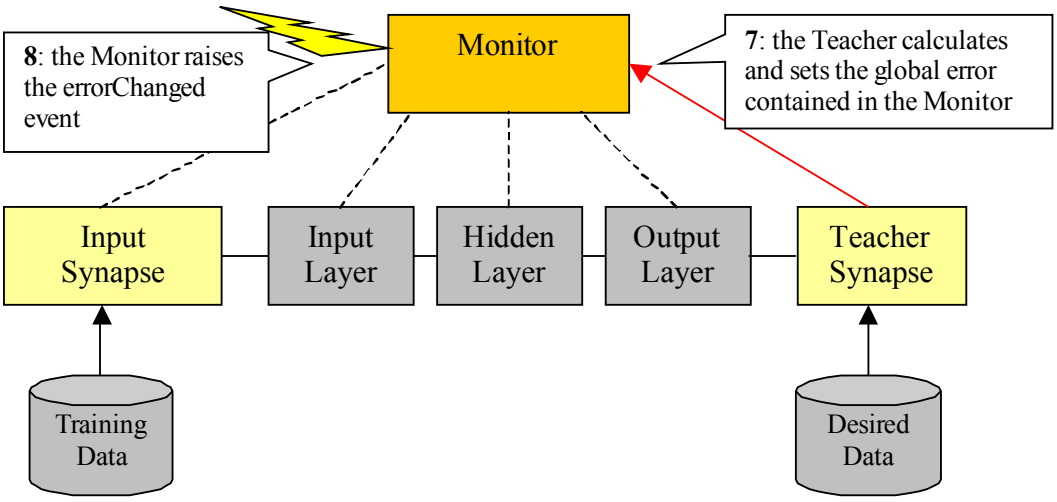
Since, as said, the last epoch is finished (i.e. the last pattern of the last cycle has been elaborated), the Monitor object raises a *netStopped* event (3) and returns a **false** Boolean value to the InputSynapse (4).

The InputSynapse, because receives a **false** value, creates a 'stop pattern' composed of a Pattern object with the counter set to –1, and injects it in the neural network (5).



All the layers of the net stop their threads – simply exiting from the run() method – when they receive a 'stop pattern' (6).

The resulting behaviour is that the neural network is stopped, and no more patterns are elaborated.



When the stop pattern reaches the TeacherSynapse, it calculates the global error and communicates this value to the Monitor object (7), which raises an *errorChanged* event to its listeners (8).

Warning: As explained in the above process, the netStopped event raised by the Monitor cannot be used to read the last error value of the net, nor to read the resulting output pattern from a recall phase, because this event could be raised when the last input pattern is still travelling across the layers, before it reaches the last output layer of the neural network.

So, to be sure to read the right values from the net, the rules explained below must be followed:

Reading the RMSE: to read the last rmse of the neural network, the errorChanged event must be waited for, so a neural network listener must be built, so the last error of the training cycle can be read and elaborated at the end of the elaboration.

Reading the outcome: to be sure to have received all the resulting patterns of a cycle from a recall phase, a 'stop pattern' must be waited for from the output layer of the net. To do this, an object belonging to the I/O components family must be built, and the code to manage the output pattern must be written into it.
Appropriate actions can be taken by checking the 'count' parameter of the received Pattern. Some pre-built output synapse classes are provided with Joone, and many others will be released in future versions.

However, as described in the next chapters, a neural network must be always used by instantiating a NeuralNet object, that hides all these internal mechanisms, and provides the user with several useful features to start/stop a neural network and to access to its internal parameters in a safe manner.

3.4.4 How the patterns and the internal weights are represented

3.4.4.1 The Pattern

The Pattern object is the ‘container’ of the data used to interrogate or train a neural network.

It is composed of two parameters: an array of doubles to contain the values of the transported pattern, and an integer to contain the sequence number of that pattern (the counter).
The dimensions of the array are set according to the dimensions of the pattern transported.

The Pattern object is also used to ‘stop’ all the Layers in the neural network. When its ‘count’ parameter contains the value –1, all the layers that will receive that pattern will exit from their ‘running’ state and will stop (the unique safe way to stop a thread in Java is to exit from its ‘run’ method). Using this simple mechanism the threads within which the Layer objects run can easily be controlled.

3.4.4.2 The Matrix

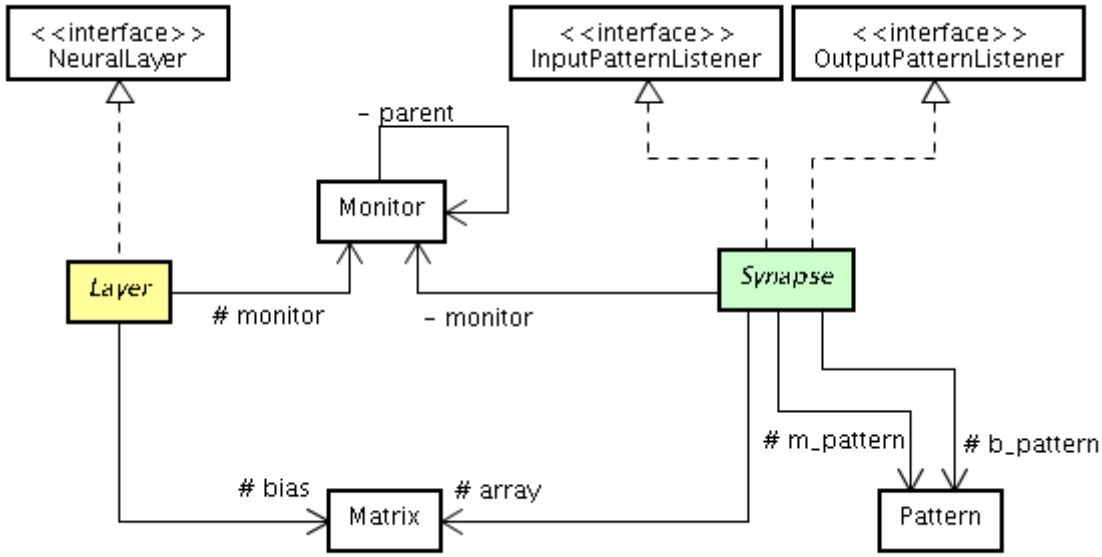
The matrix object simply contains a matrix of doubles to store the values of the weights of the connections and the biases. An instance of a matrix object is contained within both the Synapse (weights) and Layer (biases) components.

Each element of a matrix contains two values: the actual value of the represented weight, and the corresponding delta value. The delta value is the difference between the actual value and the value of the previous cycle.

The delta value is useful during the learning phase, permitting the application of momentum to quickly find the best minimum of the error surface. The momentum algorithm adds the previous variation to the actual calculated weight’s value. See the literature for more information about the algorithm.

3.5 Technical details

The core engine of Joone is composed of a small number of interfaces and abstract classes forming a nucleus of objects that implement the basic behaviours of a neural network illustrated in the previous chapter.
The following UML class diagram contains the main objects constituting the model of the core engine of Joone:



All the objects implement the *java.io.Serializable* interface, so each neural network built with Joone can be saved as a byte stream to be stored in a file system or data base, or be transported to other machines to be used remotely.
The two main components are represented by two abstract classes (both contained in the *org.joone.engine* package): the **Layer** and the **Synapse** objects.

3.5.1 The Layer abstract class

The Layer object is the basic element that forms the neural net.
It is composed of neurons, all having the same characteristics. This component transfers the input pattern to the output pattern by executing a transfer function. The output pattern is sent to a vector of Synapse objects attached to the layer's output. It is the active element of a neural net in Joone, in fact it runs in a separated thread (it implements the *java.lang.Runnable* interface) so that it can run independently from other layers in the neural net.
Its heart is represented by the method *run*:

```
    public void run() {
        while (running) {
            int dimI = getRows();
```

```
            int dimO = getDimension();
            // Recall phase
            inps = new double[dimI];
             this.fireFwdGet();
             if (m_pattern != null) {
                 forward(inps);
                 m_pattern.setArray(outs);
                 fireFwdPut(m_pattern);
             }
            if (step != -1)
                // Checks if the next step is a learning step
                m_learning = monitor.isLearningCicle(step);
            else
                // Stops the net
                running = false;
              // Learning phase
            if ((m_learning) && (running)) {
                gradientInps = new double[dimO];
                 this.fireRevGet();
                 backward(gradientInps);
                 m_pattern = new Pattern(gradientOuts);
                 m_pattern.setCount(step);
                 fireRevPut(m_pattern);
            }
        }  // END while (running = false)
        myThread = null;
    }
```

The end of the cycle is controlled by the *running* variable, so the code loops until some ending event occurs.
The two main sections of the code have been highlighted with a border:

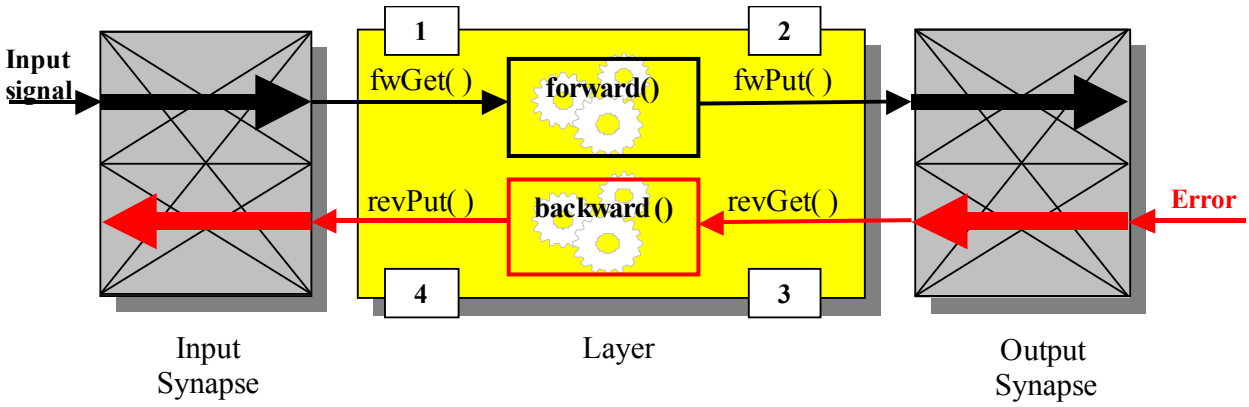
3.5.1.1 The Recall Phase

The code in the first block reads all the input patterns from the input synapses (`fireFwdGet`), where each input pattern is added to the others to produce the *inps* vector of doubles. It then calls the `Forward` method, which is an abstract method in the Layer object. In the forward method the inherited classes must implement the required formulas of the transfer function, reading the input values from the *inps* vector and returning the result in the *outs* vector of doubles. By using this mechanism based on the *template pattern*, new kind of layer can easily be built by extending the Layer object.
After this, the code calls the `fireFwdPut` method to write the calculated pattern to the output synapses, from which subsequent layers can process the results in the same manner.
In more simple terms the layer object's behaviour acts like a pump that decants the liquid (the pattern) from one recipient (the synapse) to another.

3.5.1.2 The Learning Phase

After the recall phase, if the neural net is in a training cycle, the code calls the `fireRevGet` method to read the error obtained on the last pattern from the output synapses, then calls the abstract `backward` method where, like in the forward method, the inherited classes must implement the processing of the error to modify the biases of the neurons constituting the layer. The code does

this task by reading the error pattern in the gradientInps vector and writing the result to the gradientOuts vector.
After this, the code writes the error pattern contained in the gradientOuts vector to the input synapses (`fireRevPut`), from which other layers can subsequently process the back propagated error signal.
To summarize the concepts described above, the Layer object alternately 'pumps' the input signal from the input synapses to the output synapses, and the error pattern from the output synapses to the input synapses, as depicted in the following figure (the numbers indicate the sequence of the execution):



3.5.2 Connecting a Synapse to a Layer

To connect a synapse to a layer, the program must call the `Layer.addInputSynapse` method for an input synapse, or the `Layer.addOutputSynapse` method for an output synapse.
These two methods, inherited from the `NeuralLayer` interface, are implemented in the Layer object as follows:

```
    /** Adds a new input synapse to the layer
     * @param newListener neural.engine.InputPatternListner
     */
    public synchronized void addInputSynapse(InputPatternListener newListener) {
        if (aInputPatternListener == null) {
            aInputPatternListener = new java.util.Vector();
        };
        aInputPatternListener.addElement(newListener);
        if (newListener.getMonitor() == null)
            newListener.setMonitor(getMonitor());
        this.setInputDimension(newListener);
        notifyAll();
    }
```

The Layer object has two vectors containing the list of the input synapses and the list of the output synapses connected to it.
In the `fireFwGet` and `fireRevPut` methods the Layer scans the input vector and, for each input synapse found, it calls the `fwGet` and the `revPut` methods respectively (implemented by the input synapse from the `InputPatternListener` interface).
Look at the following code that implements the fireFwGet method:

```
    /**
```

```
     * Calls all the fwdGet methods on the input synapses in order
     * to get the input patterns
     */
    protected synchronized void fireFwdGet() {
        double[] patt;
        int currentSize = aInputPatternListener.size();
        InputPatternListener tempListener = null;
         for (int index = 0; index < currentSize; index++){
             tempListener =
                    (InputPatternListener)aInputPatternListener.elementAt(index);
             if (tempListener != null) {
                 m_pattern = tempListener.fwdGet();
                 if (m_pattern != null) {
                     patt = m_pattern.getArray();
                     if (patt.length != inps.length)
                         inps = new double[patt.length];
                     sumInput(patt);
                     step = m_pattern.getCount();
                 }
             };
         };
    }
```

In the bordered code there is a loop that scans the vector of input synapses.
The same mechanism exists for the `fireFwPut` and `fireRevGet` methods applied to the vector of output synapses implementing the `OutputPatternListener` interface.
This mechanism is derived from the *Observer Design Pattern*, where the Layer is the *Subject* and the Synapse is the *Observer*.
Using these two vectors, it is possible to connect many synapses (both input and output) to a Layer, permitting complex neural net architectures to be built.

3.5.3 The Synapse abstract class

The Synapse object represents the connection between two layers, permitting a pattern to be passed from one layer to another.
The Synapse is also the 'memory' of a neural network. During the training process the weighs of the synapse (contained in the Matrix object) are modified according the implemented learning algorithm.
As described above, a synapse is both the output synapse of a layer and the input synapse of the next connected layer in the NN. To do this, the synapse object implements the `InputPatternListener` and the `OutputPatternListener` interfaces.
These interfaces contain respectively the described methods `fwGet`, `revPut`, `fwPut` and `revGet`.

The following code describes how they are implemented in the Synapse object:

```
    public synchronized void fwdPut(Pattern pattern) {
        if (isEnabled()) {
            count = pattern.getCount();
            if ((count > ignoreBefore) || (count == -1)) {
                while (items > 0) {
                    try {
                        wait();
                    } catch (InterruptedException e) {
```

```
                        return; }
                }
                m_pattern = pattern;
                inps = (double[])pattern.getArray();
                forward(inps);
                ++items;
                notifyAll();
            }
        }
    }
    public synchronized Pattern fwdGet() {
        if (!isEnabled())
            return null;
        while (items == 0) {
            try {
                wait();
            } catch (InterruptedException e) {
                return null;
            }
        }
        --items;
        notifyAll();
        m_pattern.setArray(outs);
        return m_pattern;
    }
```

The Synapse is the shared resource between two Layers that, as already mentioned, in the multi-thread mode run on two separate threads. To avoid a layer trying to read the pattern from its input synapse before the other layer has written it, the shared synapse in synchronized.

Looking at the code, the variable called ‘`items`’ represents the semaphore of this synchronization mechanism. After the first Layers calls the `fwdPut` method, the items variable is incremented to indicate that the synapse is ‘full’. Conversely, after the subsequent Layer calls the `fwdGet` method, this variable is decremented, indicating that the synapse is ‘empty’.

Both the above methods control the ‘items’ variable when they are invoked:

1. If a layer tries to call the `fwPut` method when `items` is greater then zero, its thread falls in the wait state, because the synapse is already full.
2. In the `fwGet` method, if a Layer tries to get a pattern when `items` is equal to zero (meaning that the synapse does not contain a pattern) then its corresponding thread falls in the wait state.

The `notifyAll` call at the end of the two methods permits the ‘awakening’ of the other waiting layer, signalling that the synapse is ready to be read or written. After the `notifyAll`, at the end of the method, the running thread releases the owned object permitting another waiting thread to take ownership. Note that although all waiting threads are notified by `notifyAll`, only one will acquire a lock and the other threads will return to a wait state.
The synchronizing mechanism is the same in the corresponding `revGet` and `revPut` methods for the training phase of the neural network.

The `fwPut` method calls the abstract `forward` method (at the same time as the `revPut` calls the abstract `backward` method) to permit to the inherited classes to implement respectively the recall and the learning formulas, as already described for the Layer object (according to the *Template design pattern*).
By writing the appropriate code in these two methods, the engine can be extended with new synapses and layers implementing whatever learning algorithm and architecture is required.

Starting from Joone v.2.0, in the new single-thread engine the mechanism is very similar, but in this case the xxxGet/Put methods are invoked by a single thread (instantiated by the NeuralNet object), so there isn't any conflict between concurrent threads, resulting in an improvement of the performances of about 50%.

4 I/O components: a link with the external world

The I/O components of the core engine implement the mechanism needed to make possible the connection of a neural network to external sources of data, either to read the patterns to elaborate, or to store of the results of the network to whatever output device is required.
All the I/O components extend the Synapse object, so they can be 'attached' to the input or the output of a generic Layer object since they expose the same interface required by any i/o listener of a Layer.
Using this simple mechanism the Layer is not affected by the kind of synapse connected to it because as they all have the same interface, the Layer will continue to call the Get and Put methods without needing to know more about their specialization.

4.1 The Input mechanism

To permit the user to utilize any source of data as input of a neural network, a complete input mechanism has been designed into the core engine.
The main concept underlying the input system is that a neural network elaborates 'patterns'. A pattern is composed by a row of values [x_{I1}, x_{I2}, ..., x_{IN}] representing an instance of the input dataset.
The neural network reads and elaborates sequentially all the input *rows* (all constituted by the same number of values – or *columns*) and for each one it generates an output pattern representing the outcome of the entire process.
We need two main features to reach the goal to make this mechanism as more as flexible we can:

Firstly, to represent a row of values Joone uses an array of double, hence to permit to use whatever format of data from whatever source, we need a '**format converter**'. It's based on the concept that a neural network can elaborate only numerical data (integer or real), hence a system to convert any external format to numeric values is provided. This acts as a 'pluggable' driver: with Joone is provided an interface and some basic drivers (for instance one to read ASCII values and another to read Excel sheets) to convert the input values to an array of double - the unique format accepted by a neural network to work properly.
This mechanism is expansible, as everyone can write new drivers implementing the provided interfaces.

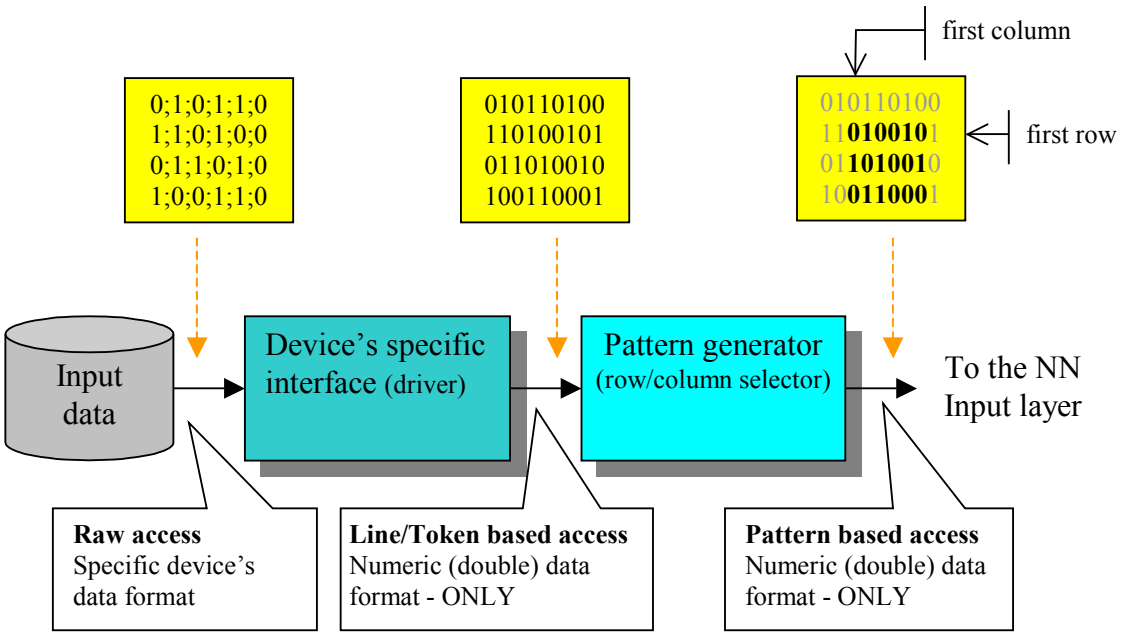
Secondly, because normally not all the available rows and columns have to be used as input data, a '**selection mechanism**' to select the input values is provided. This second feature is implemented as a component interposed between the above driver and the first layer of the neural network.

The 'cut' of the needed input columns is made by using a paremeter named `AdvancedColumnSelector`.
The advanced column selector specifies what columns from the input source should be presented the next layer. For example if a file input contains 5 columns, you could specify that only columns 1 and 3 be presented to the next layer. The selector must be a list of one or a comma delimeted list of multiple options. The options can be one column '2' or a range of columns '3-6'. The format for the selector is as follows ...

```
[col]{,[col]{,[col1-col2]}}
```

For example if the input source has 5 columns and you would like to use column 1 and columns 3,4 and 5, you could specify the selector as '1,3-5' or '1,3,4,5'.
For specific needs the same column can be read many times within the same pattern, simply specifying the same number more than once, like in the following example: '1,3,3,3,4'. A complete example of this feature is illustrated in the Chapter 9.

The overall input system is depicted in the following figure:



Note that the component connected to the first layer of the neural network, implemented within the `StreamInputSynapse` class, is built like a synapse, as it implements the corresponding interface, so the input layer is not bothered about the kind of synapse attached to it.

This is one of the most important characteristics of Joone, permitting to build whatever architecture simply gluing together several components.
The StreamInputSynapse class exposes several other parameters, other than the already seen AdvancedColumnSelector, that are inherited by all the xxxInputSynapses that extend the above abstract class:

- **Name** The name of the input synapse. It's always a good norm to set a name for each input synapse, in order to be able to manage them when attached to other I/O components (like, for instance, the InputSwitchSynapse).
- **Buffered** Determines whether the data should be buffered in memory rather than being read throughout the run. By default all the input synapses are buffered. This is because the buffering is a useful feature when a neural network has to be transported to another machine for remote training. If the input synapses are buffered, all the input data are transported along with the neural network, avoiding to retrieve them remotely.
- **FirstRow** The first row of the file that contains useful information.
- **LastRow** The last row of the file that contains useful information. Default of zero uses all the rows.
- **Enabled** The component is working only when this property is true.
- **StepCounter** Input layers affect the running of the network. By default, each time a line is read from an input layer, the network monitor advances one step in the learning process. **Note**: If there are several input layers, only one should have the step counter enabled.
- **MaxBufSize** Indicates the max buffer's size used to store the input patterns. If equal to 0 (the default), the buffer size is set to 1MB (augment it only if your input data source exceeds such size). Used only if 'Buffered' = true, otherwise it is ignored. Must be equal or greater than the size of the input buffer expressed in bytes.

4.1.1 The FileInputSynapse

A file input synapse allows data to be presented to the network from a file. The file must contain columns of integer or real values delimited by a semicolon.

E.g for the xor problem the file should contain...

```
0;0;0
1;0;1
0;1;1
1;1;0
```

There is an extra property that can be set for file input layers:

- **FileName**
  The name of the file containing the data. E.g. c:\data\myFile.txt

4.1.2 The URLInputSynapse

This component allows data to be presented to the network from a URL.

The protocols supported are HTTP and FTP. The file pointed by the URL must contain numbers separated by a semicolon, the same format accepted by the FileInputSynapse.

There extra property that should be set for URL input layers:

- **URL**
  The name of the Unified Resource Locator containing the data. E.g. http//www.someServer.org/somepath/myData.txt or ftp://ftp.someServer.org/somePath/myData.txt

4.1.3 The ExcelInputSynapse

The Excel Input synapse permits data from an Excel file to be applied to a neural network for processing.

The extra properties that can be specified for Excel input layers:

- **fileName**
  The parameter allows the name of the sheet to be chosen from which the input data is read.
- **Sheet**
  The parameter allows the name of the sheet to be chosen from which the input data is read. If blank, the first available sheet is used.

4.1.4 The JDBCInputSynapse

The JDBCInputSynapse permits data from almost any database to be applied to a neural network for processing.

To use this input synapse you should ensure that the required JDBC Type 4 Driver is in the class path of your application. It is possible to use other JDBC driver types though you will have to refer to the vendors documentation, it may require extra software installation and this may limit your distribution to certain Operating Systems.

The extra properties that can be specified for JDBC input layers:

- **driverName**
  The name of the database driver. For example if you were using the JdbcOdbc driver provided by Sun and already present in the java distribution then 'sun.jdbc.odbc.JdbcOdbcDriver'
- **dbURL**
  The database specification. This protocol is specific to the driver, you must check the protocol with the driver vendor. For example for the JdbcOdbc bridge 'jdbc:mysql://localhost/MyDb?user=myuser&password=mypass'
- **SQLQuery**
  The query that you will use to extract information from the database. For example '`select input1,input2,output from xortable;`'

Some commonly used Driver protocols are shown below ...

Driver {com.mysql.jdbc.Driver}

Protocol
{jdbc:mysql://[hostname][,failoverhost...][:port]/[dbname][?param1=value1][&param2=value2].....} MySQL Protocol

Example {jdbc:mysql://localhost/test?user=blah&password=blah}

See http://www.mysql.com

Driver {sun.jdbc.odbc.JdbcOdbcDriver}

Protocol { jdbc:odbc:[;=]* } ODBC Protocol

Example {jdbc:odbc:mydb;UID=me;PWD=secret}

See http://www.java.sun.com

Data Types

Any fields selected from a database should contain a single integer, double or float format value. The data type is not so important it can be text or a number field so long as it contains just one integer, double or float format value.

E.g Correct = '2.31' Correct = '-15' Wrong= '3.45;1.21' and Wrong = 'hello'

4.1.5 The ImageInputSynapse

This input synapse collects data from Image files or Image objects and feeds the data from the Images into the Neural network. Images can be read either from the file system, or from a predefined array of Images.
GIF, JPG and PNG image file formats can be read.
The synapse operates in two modes, colour and grey scale.

Colour Mode
In colour mode ImageInputSynapse produces seperate RGB input values in the range 0 to 1 from the image. So using an image of width 10 and height 10 there will be 10x10x3 inputs in the range 0 to 1.
The individual colour components are calculated by obtaining the RGB values from the image. These value are initially in an ARGB format. Transparency is removed and the RGB value extracted and normalized between 0 and 1.

Non Colour Mode / Grey Scale Mode
In this mode the synapse treats each input value as a grey scale value for each pixel. In this mode only Width*Height values are required. To produce the final image the Red, Green and Blue components are set to this same value.
The grey scale component is calculated by obtaining the RGB values from the image. These values are initially in an ARGB format. Transparency is removed and the RGB value extracted, averaged and normalised to produce one grey scale value between 0 and 1.

The following properties must be provided to this synapse...

- **ImageDirectory**
  This is the path to the directory that contains the images to elaborate. By default it's equal to the value of the “user.dir” system property.
- **FileFilter**
  A regex containing the filter used to read the image files from the file system. By default equal to the string ".*[jJ][pP][gG]" (.jpg and .JPG files)
- **ImageInput**
  (optional) Points to an array of Image objects. If this property is used, the images will be read from the array, and the above two properties will be ignored.
- **DesiredWidth/DesiredHeight**
  These two properties indicate the desired size of the images that will be used to feed the neural network. All the images will be rescaled in order to respect the indicated size (by default both the dimensions are set to 10 pixel)
- **ColourMode**
  A boolean value indicating the operating mode (see above). By default the synapse will operate in colour mode (ColourMode=true).

4.1.6 The YahooFinanceInputSynapse

The YahooFinanceInputSynapse provides support for financial data input from financial markets. The synapse contacts Yahoo Finance services and downloads historical data for the chosen symbol and date range. Finally the data is presented to the network in reverse date order i.e oldest first.

The following properties must be provided to this synapse...

- **Symbol**
  This is the symbol of the specific stock e.g TSCO.L for UK super market company Tesco's. This must be one of symbols defined by Yahoo.

- **firstDate**
  This is the date of the oldest requested stock value. Note the dates should be in the following format YYYY.MM.DD where YYYY=4 Character year, MM=2 character month, DD=2 character day of the month.

- **lastDate**
  This is the date of the latest requested stock value. This uses the same format as First Date above.

- **Period**
  This is the period between stock values obtained from Yahoo, '**Daily**' will obtain stock values recorded at the end of each day, '**Monthly**' will obtain stock values recorded at the beginning of each month, '**Yearly**' will obtain stock values recorded at the start of each year.

This synapse provides the following info:

Open as column 1
High as column 2
Low as column 3

Close as column 4.
Volume as column 5.
Adjusted Close as column 6

For the particular stock symbol. You must set the Advanced Column Selector (ACS) according to this values. If you want to use as input the Open, High and the Volume columns, hence you must write '1-2,5' into the ACS.
Note the stock symbol must be one of the symbols defined by Yahoo. For a list of symbols see the Finance section of the Yahoo web site http://finance.yahoo.com.

4.1.7 The MemoryInputSynapse

The memoryInputSynapse allows data to be presented to the network from an array of double. This is a component very useful to use when an external program needs to feed the network with data got from external or internal sources for which a specific xxxInputSynapse doesn't exist.

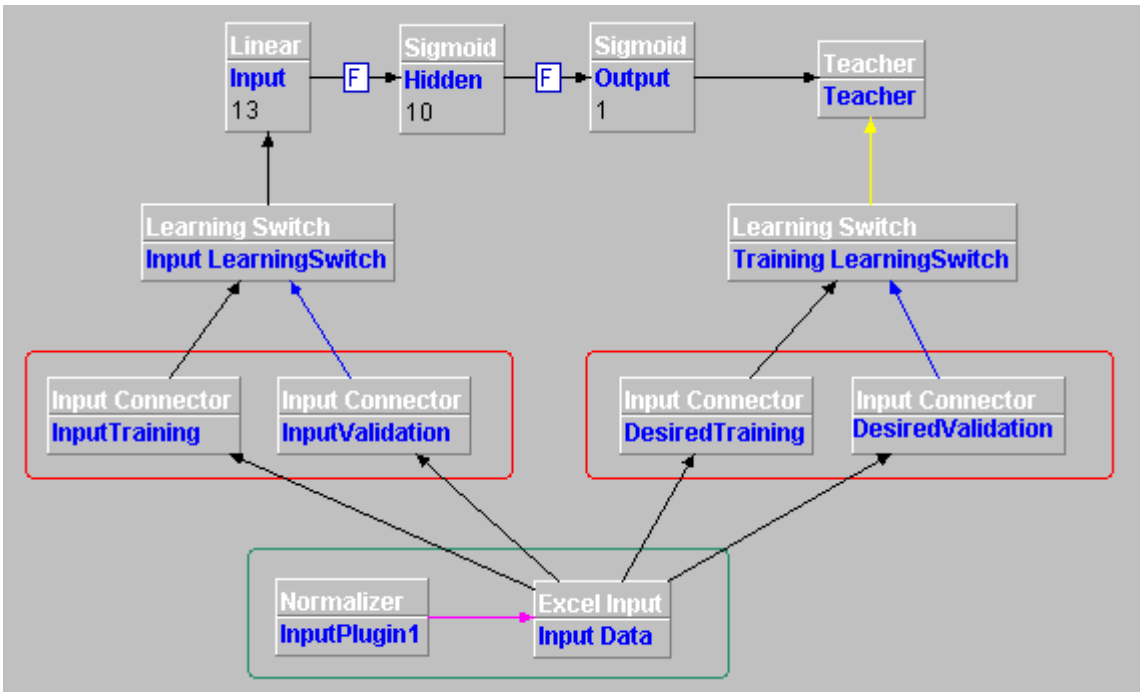
The following property must be provided to this synapse...

- **inputArray**
  This property must contain a pointer to the double[][] array containing the input data.

4.1.8 The InputConnector

When we need to train a network, we need to use at least two input data sources, one as training input data and another as desired data. If we add also another data source to validate the network, then we need to add other two data sources to our neural network. All those input synapses make the architecture of the neural network very complex, and, if they are buffered, a huge amount of memory is needed in order to store all the implied data.

In order to resolve the above problems, we have built a new input component named InputConnector. It permits to share the same input synapses for several uses, as depicted in the following figure (created by a snapshot of the drawing area of the GUI Editor):



The InputConnectors are into the two red boxes, in the above figure.
As you can see, thanks to the InputConnector components, we have used only one input data source (the ExcelInputSynapse), and only one NormalizerPlugin, simplifying, in this manner, the entire architecture of the neural network.

The four InputConnectors are used to read from the Excel sheet, respectively:
- The input data for the training phase (InputTraining connector)
- The input data for the validation phase (InputValidation connector)
- The desired data for the training phase (DesiredTraining component)
- The desired data for the validation phase (DesiredValidation component)

In order to see how we have used them, we need to list the main properties they expose:

- **advancedColumnSelector**
  contains the columns to read from the connected input synapse
- **first/lastRow**
  they contain the first and the last row we want to read from the input synapse
- **buffered**
  if true, the InputConnector reads all the patterns from the connected input synapse when the network starts, and stores them within an internal buffer (by default this property is set to false). **Note**: set to true ONLY when an input plugin is connected to the InputConnector (read below)

the above properties are set independently from the corresponding ones of the connected input synapse.
In the above figure we could have the following settings (in this example we'll use the first 100 rows for training and the next 50 for validation; the first 13

columns are the independent variables and the col. 14 contains the target value):

The settings for the ExcelInputSynapse:

```
ExcelInputSynapse.advancedColumnSelector = "1-14"
ExcelInputSynapse.firstRow = 1
ExcelInputSynapse.lastRow = 0
ExcelInputSynapse.buffered = true
```

settings for the InputConnector named 'InputTraining':

```
InputTraining.advancedColumnSelector = "1-13"
InputTraining.firstRow = 1
InputTraining.lastRow = 100
InputTraining.buffered = false
```

...the InputConnector named 'InputValidation':

```
InputValidation.advancedColumnSelector = "1-13"
InputValidation.firstRow = 101
InputValidation.lastRow = 150
InputValidation.buffered = false
```

...the InputConnector named 'DesiredTraining':

```
DesiredTraining.advancedColumnSelector = "14"
DesiredTraining.firstRow = 1
DesiredTraining.lastRow = 100
DesiredTraining.buffered = false
```

...and the InputConnector named 'DesiredValidation':

```
DesiredValidation.advancedColumnSelector = "14"
DesiredValidation.firstRow = 101
DesiredValidation.lastRow = 150
DesiredValidation.buffered = false
```

As illustrated in the example, the ExcelInputSynapse is buffered, and contains all the rows and all the columns needed, while each single InputConnector reads only the 'piece' of input data it needs, according to its position and purpose within the neural network.
The four InputConnectors are all unbuffered, so we occupy only the strictly necessary memory of our machine.

The 'buffered' property of the InputConnector class exists because we could have to set it to true in case we need to pre-process the data of a particular InputConnector using an InputPlugin, and this is because the input plugins work only for buffered synapses. In this manner we have the maximum flexibility, being able to 'cut' the input data as we want, and also pre-process separately each single piece of data, if necessary.
Of course you must use a buffered InputConnector only when really necessary, in order to avoid to waste valuable memory.

It's very simple to use the InputConnector class in a java program. Here is a little example:

```
// Create the InputSynapse
XLSInputSynapse inputSynapse = new XLSInputSynapse();
inputSynapse.setFileName("myData.xls");
inputSynapse.setAdvancedColumnSelector("1-14");
...
// Create the InputConnector
InputConnector inputTraining = new InputConnector();
inputTraining.setAdvancedColumnSelector("1-13");
...
// Connect the InputSynapse to the InputConnector
inputTraining.setInputSynapse(inputSynapse);
// Connect the InputConnector to the input layer of the network
LinearLayer inputLayer = new LinearLayer();
inputLayer.addInputSynapse(inputTraining);
...
```

4.2 The Output: using the outcome of a neural network

The `Output components` allow a neural network to write output patterns to a whatever storing support.
They write all the values of the pattern passed by the calling attached Layer to an output stream, permitting the output patterns from an interrogation phase to be written as, for example, ASCII files, FTP sites, spreadsheets, charting visual components, etc.

Joone has several real implementations of the output classes to write patterns in the following formats:

- Comma separated ASCII values
- Excel spreadsheets
- Images (JPG)
- RDBMS tables using JDBC
- Java Arrays - to write the output in a 2D array of doubles, in order to use the output of a neural network from an embedding or external application.

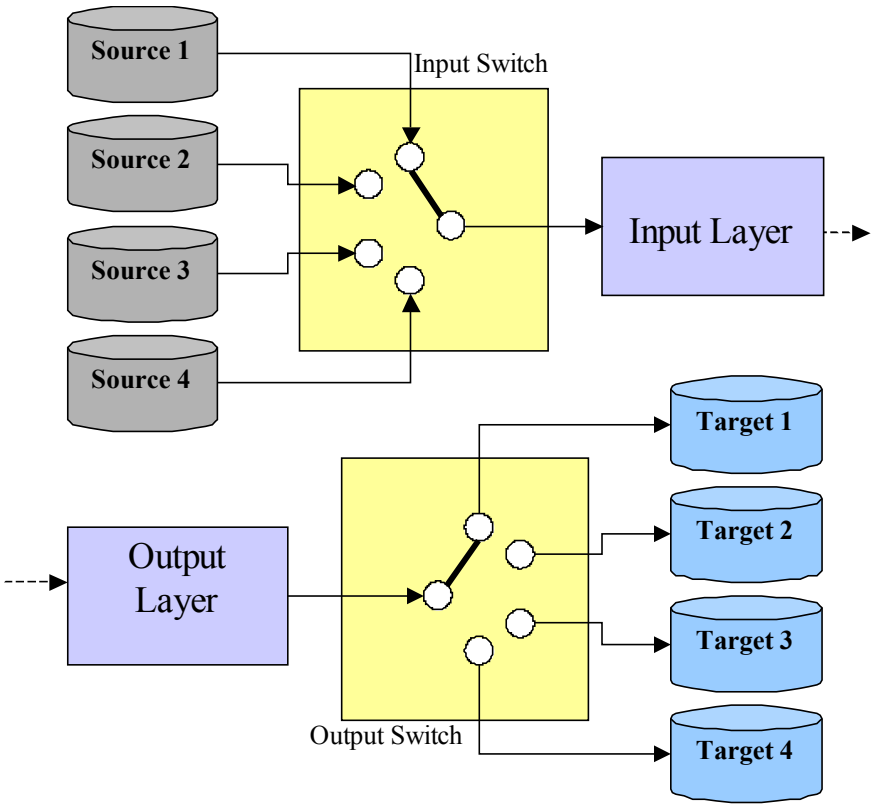
In the Chapter 9 some techniques to get and use the outcome of a neural network will be shown.

Many others output components can be added simply extending the basic abstract classes provided with the core engine; in this manner Joone could be used to manipulate several physical devices like robots arms servomotors, regulator valves, servomechanisms, etc.

4.3 The Switching Mechanism

Sometime it is useful to change the input source of a neural network depending on a network's state or on some event. For example it might be the necessity to test a trained neural network on several input patterns, or to train a net using input patterns coming from several sources.
The same idea would also be useful on the output of a neural network because the user might need to dynamically change the destination of the network output stream.

This mechanism is shown in the following figure:



Joone has a mechanism to dynamically change the input source and the output destination of a neural network. This is based on two components: the `Input Switch and the Output Switch.`

4.3.1 The InputSwitch

Any InputSynapse (any object capable to read external source of data) can be attached to this component. As it acts as a switch, the active input source (i.e. the input source attached to the neural network) can be changed dynamically simply by indicating the name of the input synapse that is to be made the active input of the neural network.

4.3.2 The OutputSwitchSynapse

Any Output synapse can be attached to this component. As it acts as an output switch, the active output target can be dynamically changed simply by

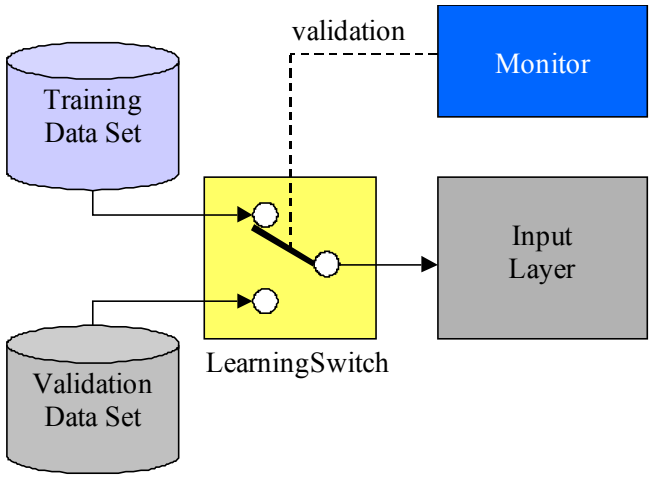
indicating the name of the output synapse that is to be made the active output of the neural network.

4.4 The Validation Mechanism

Validating a neural network during its training cycles is very useful to determine the generalization capability of the net. This verification is made by measuring the error of the net using a set of patterns that have not been used by the net during the training cycles.

It is a good rule to reserve a certain number of rows of the training patterns to execute the validation check. The following outlines how this would be done with a neural network built with Joone.
First of all, a mechanism is required to automatically switch between the training and the validation data sets. To do this, an extension of the Input Switch has been built.

The following schema illustrates the required architecture:



The `LearningSwitch` can change its state according to the value of the `validation` parameter of the Monitor object. Hence, depending on this parameter, the switch will either connect the training or the validation data set to the input layer of the neural network.

The same schema must also be applied to the **desired** data sets, inserting a LearningSwitch between the training and validation desired data sets, and the TeachingSynapse.

After having built a neural network according to the described architecture, the validation check can be performed in the following manner:

1. The neural network is trained for a certain number of cycles.

2. A clone of the neural network is obtained by calling the `NeuralNet.cloneNet()` method.
3. The Monitor of the cloned net is set to these values:
    a. The `totCycles` parameter is set to 1.
    b. The `validation` parameter is set to true.
4. The neural network is interrogated, and the RMSE value is measured.
5. If the RMSE value is less than a desired threshold, the training cycle is stopped, otherwise the cycle continues from the step 1.

Steps 2, 3, 4 and 5 can be performed in response to a cycleTerminated event of the trainee neural network.

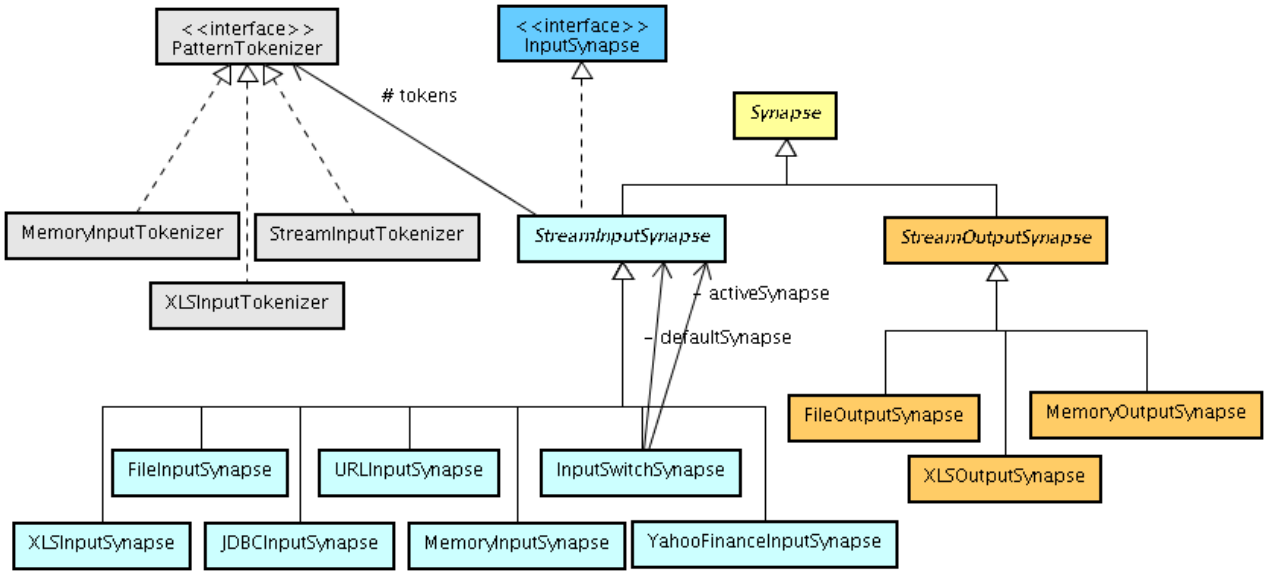
Note that it is not necessary to explicitly set the `validation` parameter of the net before the step 1 because its default value is equal to false (i.e. the training data set is connected to the input layer).

The cloning of the net in the step 2 is performed to obtain a ‘dummy’ neural network to change and use for the validation steps, without having to save and then restore the old state to correctly continue the training.

A complete example about how to implement this technique is described in the Chapter 9.

4.5 Technical details

The I/O components of the core engine are stored in the `org.joone.io` package.
They permit both the connection of a neural network to external sources of data and the storage of the results of the network to whatever output device is required.
The object model is shown in the following figure:



The abstract StreamInputSynapse and StreamOutputSynapse classes represent the core elements of the IO package.

They extend the abstract Synapse class, so they can be 'attached' to the input or the output of a generic Layer object since they expose the same interface required by any i/o listener of a Layer.
Using this simple mechanism the Layer is not affected by the category of synapses connected to it because as they all have the same interface, the Layer will continue to call the xxxGet and xxxPut methods without needing to know more about their specialization.

4.5.1 The StreamInputSynapse

The StreamInputSynapse object is designed to provide a neural network with input data by providing a simple method to manage data that is organized as rows and columns, for instance as semicolon-separated ASCII input data streams.
Each value in a row will be made available as an output of the input synapse, and the rows will be processed sequentially by successive calls to fwdGet method.

As some files may contain information additional to the required data, the parameters `firstRow`, `lastRow`, and `AdvancedColumnSelector`, derived from the `InputSynapse` interface, may be used to define the range of usable data.
The Boolean parameter `stepCounter` indicates if the object is to call the Monitor.nextStep() method for each pattern read.

By default it is set to TRUE but in some cases it must be set to FALSE.
In fact, into a neural network that is to be trained, we need to put at least two StreamInputSynapse objects: one to give the sample input patterns to the neural network and another to provide the net with the desired output patterns to implement some supervised learning algorithm.
Since the Monitor object is the same for all the components in a neural network built with Joone, there can be only one input component that calls the Monitor.nextStep() method, otherwise the counters of the Monitor object will be modified twice (or more) for each cycle.
To avoid this side effect, the stepCounter parameter of the StreamInputSynapse that provides the desired output data to the neural network, is set to FALSE.

A StreamInputSynapse can store its input data permanently by setting the `buffered` parameter to TRUE (the default). So an input component can be saved or transported along with its input data, permitting a neural network to be used without the initial input file. This feature is very useful for remotely training a neural network in a distributed environment, as provided by the Joone framework.

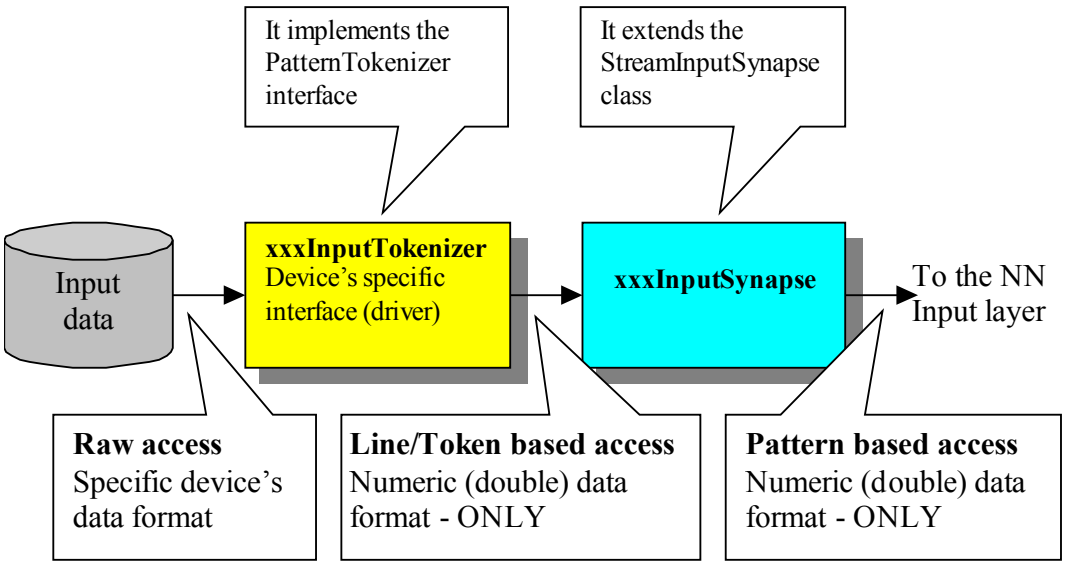
The `FileInputSynapse` and `URLInputSynapse` objects are real implementations of the abstract StreamInputSynapse class which read input patterns from files and http/ftp sockets respectively.

To extract all the values from a semicolon-separated input stream, the above two classes use the StreamInputTokenizer object. These are able to parse each line of the input data stream to extract all the single values from it and return them by the `getTokenAt` and `getTokensArray` methods.

To better understand the concepts underlying the I/O model of Joone, we must considerate that the I/O component package is based on two distinct tiers to logically separate the neural network from its input data.

Since a neural network can natively process only floating point values, the I/O of Joone is based on this assumption, then if the nature of the input data is already numeric (integer or float/double), the user doesn’t need to make further format transformations on them.

The I/O object model is based on two separated levels of abstraction, like depicted in the following figure:



The two colored blocks represent the objects that must be written to add a new input data format and/or device to the neural network.

The first is the 'driver' that knows how to read the input data from the specific input device.
It converts the specific input data format to the neural network's accepted numeric double format, also exposing a line/token (e.g. row/column) based interface to provide the xxxInputSynapse with the patterns read.

The latter is the 'adapter' that reads the data provided by the xxxInputTokenizer, selects only the desired columns and encapsulates them into a Pattern object, one for each requested row.
Each call to its fwdGet() method will provide the caller with a new read Pattern.

To add a new xxxInputSynapse that reads patterns from a different kind of input data to semicolon separated values, you must:

1. Create a new class implementing the PatternTokenizer interface (e.g. xxxInputTokenizer)
2. Write all the code necessary to implement all the public methods of the inherited interface.
3. Create a new class inherited from StreamInputSynapse (e.g. xxxInputSynapse).
4. Override the abstract method initInputStream, writing the code necessary to initialise the 'token' parameter of the inherited class. To do this, you must call the method super.setToken from within initInputStream, passing the newly created xxxInputTokenizer after having initialised it. For more details see the implementation built into FileInputSynapse.

The actual implemented **StreamInputTokenizer** is an object to transform semicolon separated ASCII values to numeric double values, and it was the first implementation made because the most common format of data is contained in text files; if the input data are already contained in this ASCII format, you can just use it, without implement any transformation.

For data contained in array of doubles, (i.e. for input provided from another application), we have built the **MemoryInputTokenizer** and the **MemoryInputSynapse** classes that implement the above two layers to provide the neural network with data contained in a 2D array of doubles.
To use them, simply create a new instance of the MemoryInputSynapse and set the input array calling its **setInputArray** method, then connect it to the input layer of the neural network.

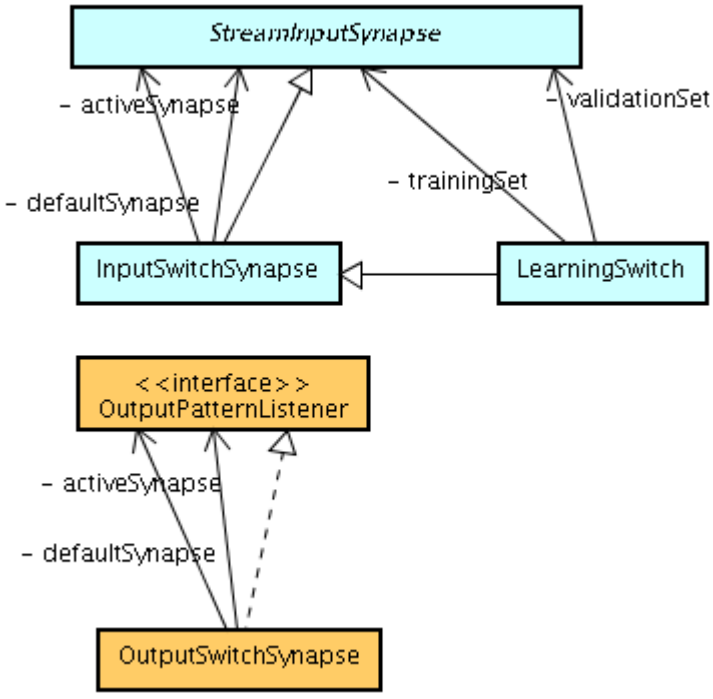
4.5.2 The StreamOutputSynapse

The `StreamOutputSynapse` object allows a neural network to write output patterns.
It writes all the values of the pattern passed by the call of fwdPut method to an output stream.

The values are written separated by the character contained in the `separator` parameter (the default is the semicolon), and each row is separated by a carriage return.
Extending this class allows output patterns from an output device to be written as, for example, ASCII files, FTP sites, spreadsheets, charting visual components, etc.

4.5.3 The Switching mechanism's object model

The following class diagram shows the corresponding object model:



4.5.3.1 The InputSwitchSynapse

Using the `addInputSynapse` method, any xxxInputSynapse (any object inheriting the StreamInputSynapse class) can be attached to this component. As it acts as a switch, the active input source can be changed dynamically simply by calling the `setActiveInput(name)` method, passing as a parameter the name of the input synapse that has to be made the active input of the neural network.
Calling the `setDefaultInput(name)` method sets the default input connected to the net.

4.5.3.2 The OutputSwitchSynapse

Using the `addOutputSynapse` method, any object inheriting the OutputPatternListener class can be attached to this component. As it acts as an output switch, the active output target can be dynamically changed simply by calling the `setActiveOutput(name)` method, passing as a parameter the name of the output synapse that has to be made the active output of the neural network.
As with the InputSwitchSynapse object, calling the `setDefaultOutput(name)` method sets the default output connected to the net.

4.5.3.3 The LearningSwitch

As described above, the LearningSwitch permits to change dynamically the input source connected to a neural network according to its validation flag.

By calling the addTrainingSet method, whatever xxxInputSynapse (any object inheriting the StreamInputSynapse class) can be attached to this component containing the training input patterns, whereas by calling the addValidationSet permits to set the xxxInputSynapse containing the validation patterns that will be used when the validation parameter is true.

5 Teaching a neural network: the supervised learning

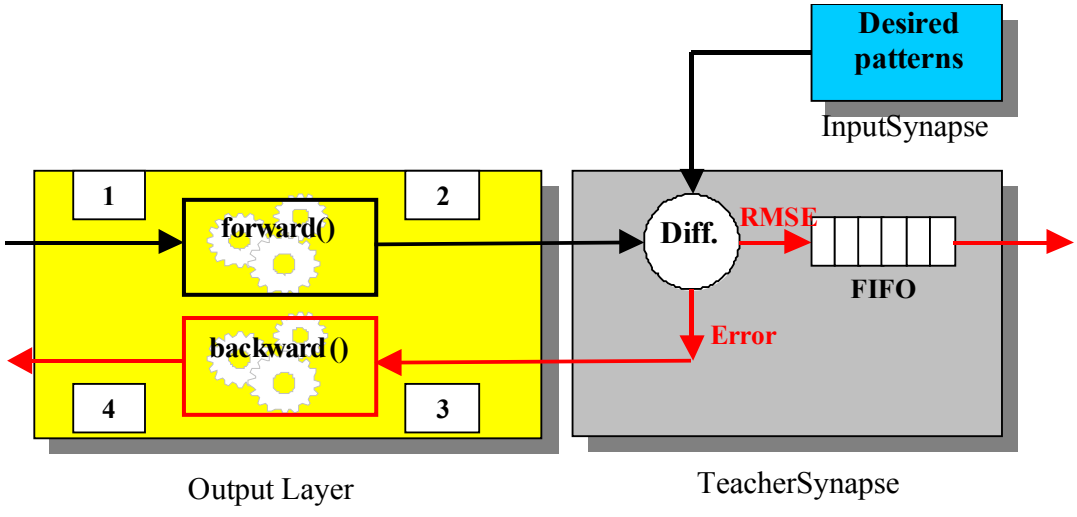
To implement the supervised learning techniques, some mechanism is needed to provide the neural network with the error for each input pattern, expressed as the difference between the output generated by the actual processed pattern and the desired output value for that pattern.

5.1 The Teacher component

The function of this component (the TeacherSynapse) is to calculate the difference between the output of the neural network and a desired value obtained from some external data source.
The calculated difference is injected backward into the neural network starting from the output layer of the net, so each component can process the error pattern to modify the internal connections by applying some learning algorithm.

The TeacherSynapse object, as its name suggests, implements the Synapse object so that it can be attached as the output synapse of the last layer in the neural network.
This basic rule, as you probably have already noticed, is a rule of thumb of all the main processing elements of Joone, permitting in this manner to easily attach each component to each other (compatibly with their nature) without to be worried about their particular specialization.

The internal composition of the Teacher object is depicted in the following figure:



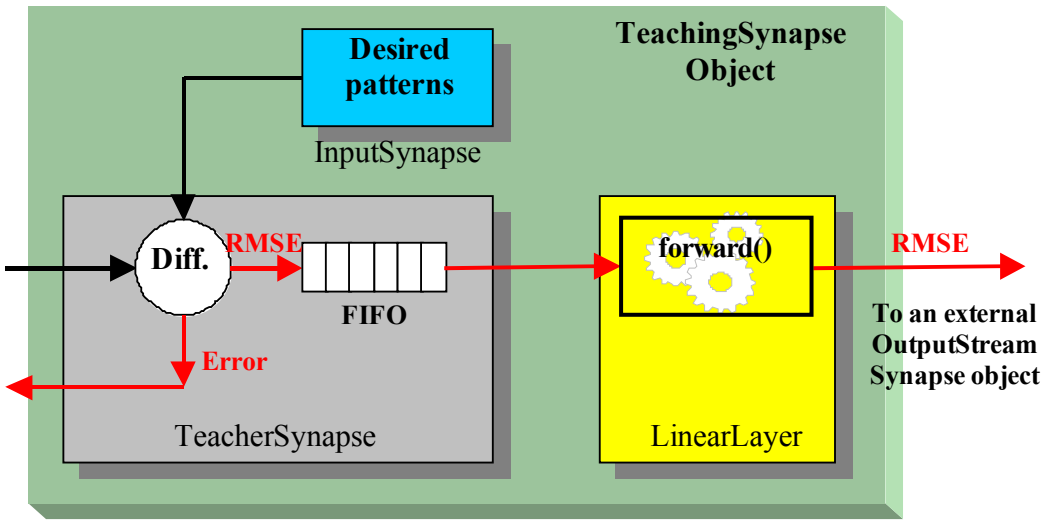
The TeacherSynapse object receives – as does any other Synapse – the pattern from the preceding Layer. The Teacher reads the desired pattern for that cycle from an InputSynapse and calculates the difference between the two patterns, making the result available to the connected Layer, which can get and inject it in the neural network to backpropagate the measured error.

So the training cycle is complete! The error pattern can be transported from the last to the first layer of the neural network using the mechanism illustrated in the previous chapters of this paper.
In this simple manner the output layer doesn't concern itself about the nature of the attached output synapse, since it continues to call the same methods known for the Synapse object.

To give to an external application the RMSE – root mean squared error - calculated on the last cycle, at the end of each cycle the TeacherSynapse pushes this value into a FIFO – First-In-First-Out - structure. From here any external application can get the resulting RMSE value in any moment during the training cycle.
The use of a FIFO structure permits loose coupling between the neural network and the external thread that reads and processes the RMSE value, avoiding the training cycles having to wait before processing of the RMSE pattern.

Note: from the version 1.2 of the core engine, the TeacherSynapse is able to calculate also the MSE (mean squared error) instead of the RMSE. This depends on the value of the boolean property useRMSE. If false (the default), the MSE is calculated.

In fact, to get the RMSE values, simply connect another Layer - that runs on a separate Thread - to the output of the TeacherSynapse object, and connect to the output of this Layer, for instance, a FileOutputSynapse object, to write the RMSE values to an ASCII file, as depicted in the following figure:



To simplify the construction of the above described chain – **`teacher -> fifo -> layer`** – a new object (called TeachingSynapse) has been built and inserted in the core engine.

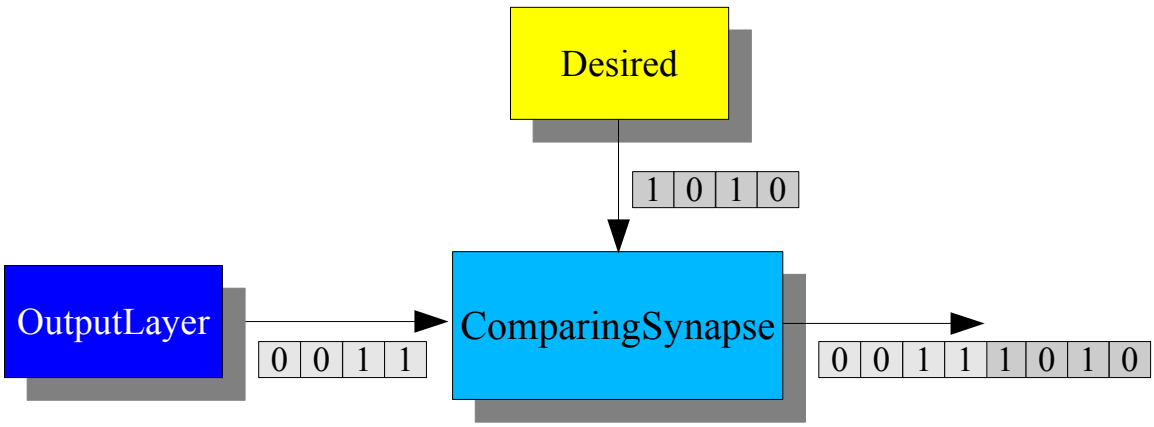
This compound object is a fundamental example about how to use the basic components of Joone to built more complex components that implement some more sophisticated feature. In other words, this is an additional example of the simplicity of the LEGO® bricks system philosophy on which Joone is based.

5.2 Comparing the desired with the output patterns

In some cases it should be useful to compare the actual output of the trainee neural network with the desired patterns used during the training phase. To do this, the ComparingSynapse has been built.

It implements the same interface of the TeachingSynapse class, so it can be used exactly like that component.

The unique difference is its output, represented by a pattern that is the composition of the two input patterns (that one coming from the output layer and the desired one), like depicted in the following figure:



As you can see, the output pattern's length is the double of that of the two inputs, and contains the composition of their content.

This component can be used to plot, for instance, the two signals into the same chart component, or can be used to write the output+desired patterns as columns of the same output file for further uses.

5.3 The Supervised Learning Algorithms

As everybody already knows, in the supervised learning, a neural network learns to resolve a problem simply by modifying its internal connections (biases

and weights) by back-propagating the difference between the current output of the neural network and the desired response.

In order to obtain that, each bias/weight of the network's components (both layers and synapses) is adjusted according to some specific algorithm.

Of course, as there isn't just one algorithm to change the internal weights of a neural network, we need a flexible mechanism in order to be able to set the training algorithm suitable for a determined problem.
Joone provides the user with several learning algorithms, and in the following paragraphs we'll see them in detail.

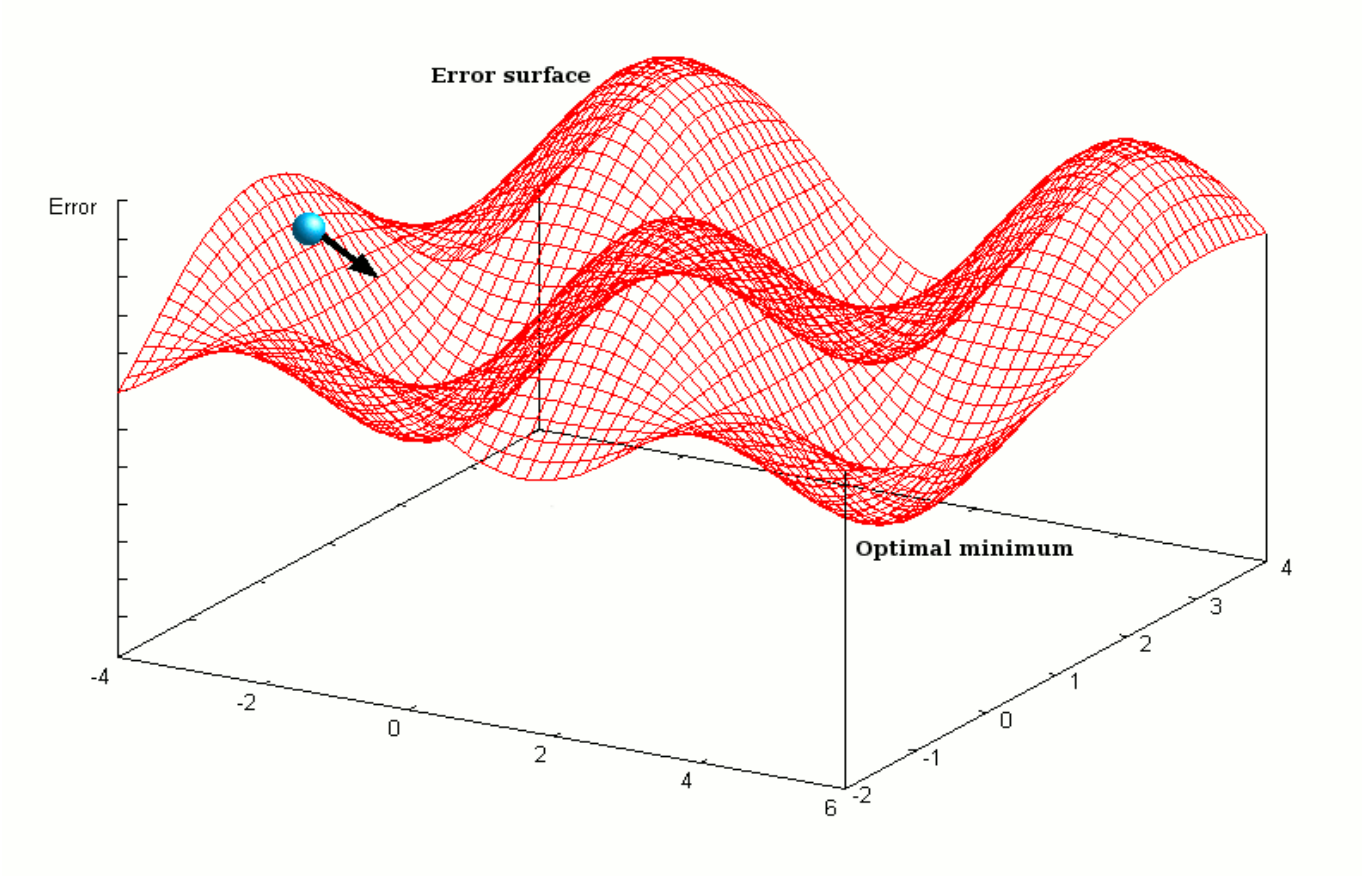
Before to continue, I want to recall that there isn't any optimal algorithm that is good for whatever problem. You need to try several of them in order to find the best one for your own specific application. For this reason Joone comes with a distributed training environment – the DTE - to permit to train in parallel mode different neural networks in order to efficiently find the best one.

5.3.1 The basic On-Line BackProp algorithm

This is the most common used training algorithm.
It adjusts the Layers' biases and the Synapses' weights according to the gradient calculated by the TeacherSynapse, and back-propagated by the backward-transportation mechanism already illustrated in the previous chapters.

It is called 'On-Line' because it adjusts the biases and weights after each input pattern is read and elaborated, so each new pattern will be elaborated using the new weights/biases calculated during the previous cycles.

The algorithm searches for a optimal combination of network's biases/weights by moving a virtual point along a multi-dimensional error surface, until a good minimum is found, like represented by the following figure (represented in three dimensions for the sake of semplicity):



The algorithm uses two parameters to work: the **learning rate**, that represents the 'speed' of the virtual point (the blue ball in the above figure) along the error surface (represented by the red grid), and the **momentum**, that represents the 'inertia' of that point.
Both these parameters must be set to a value in the range [0, +1], and good values can be found only through several trials.

Remember that, while the momentum can be set to 0, the learning rate must be always set to a value greater than zero, otherwise the network cannot learn.

5.3.2 The Batch BackProp algorithm

This is a variation of the on-line algorithm, because it works exactly like the above, except that the biases/weights adjustments are applied only at the end of each epoch (i.e. after all the input patterns of the training set have been elaborated).
It works by storing in a separate array all the changes calculated for each pattern, and applying them only at the end of each epoch.

In this manner each pattern belonging to the same epoch will be elaborated using an unmodified copy of the weights/biases. This causes more memory to be consumed by the network, but in some cases the batch algorithm converges in less epochs.

This algorithm uses, beside the same parameters of the on-line version, also another parameter named **batch size.** It indicates the number of input patterns during which we want to use the batch mode, before to apply the on-line modification of the biases/weighs.
This parameter, normally, is set to the number of training patterns, but by setting it to a smaller value, we can train our network also in mixed-mode.

5.3.3 The Resilient BackProp algorithm (RPROP)

This is an enhanced version of the batch backprop algorithm, and for several problems it converges very quickly.

It uses only the sign of the backpropagated gradient to change the biases/weights of the network, instead of the magnitude of the gradient itself.

This because, when a Sigmoid transfer function is used (characterized by the fact that its slope approaches zero as the input gets large), the gradient can have a very small magnitude, causing small changes in the weights and biases, even though the weights and biases are far from their optimal values.

Based on this modified algorithm, Rprop is generally much faster than the standard steepest descent algorithm.
As said, it is a batch training algorithm, and uses only the **batch size** property.

Note: even if the value of the learning rate and the momentum properties doesn't affect the calculus of the Rprop algorithm, you need to set the learning rate to 1.0 in order to use properly this training algorithm.

5.3.4 How to set the learning algorithm

In order to choose the needed learning algorithm of a neural network, the Monitor object exposes the getter/setter methods of the following properties:

Learners: it's an indexed list containing all the declared learners (i.e. objects implementing the `org.joone.engine.Learner` interface – see the technical details)
learningMode: it's an integer containing the index of the chosen Learner object from the above list

In order to set a learning algorithm you need to write the following java code before to start the network:

```
Monitor.getLearners().add(0, "org.joone.engine.BasicLearner"); // On-line
Monitor.getLearners().add(1, "org.joone.engine.BatchLearner"); // Batch
Monitor.getLearners().add(2, "org.joone.engine.RpropLearner"); // RPROP
Monitor.getLearners().add(3, "<whatever_else_learner_class>"); // ...

Monitor.setLearningMode(1); // We have chosen the Batch learning in this case
```

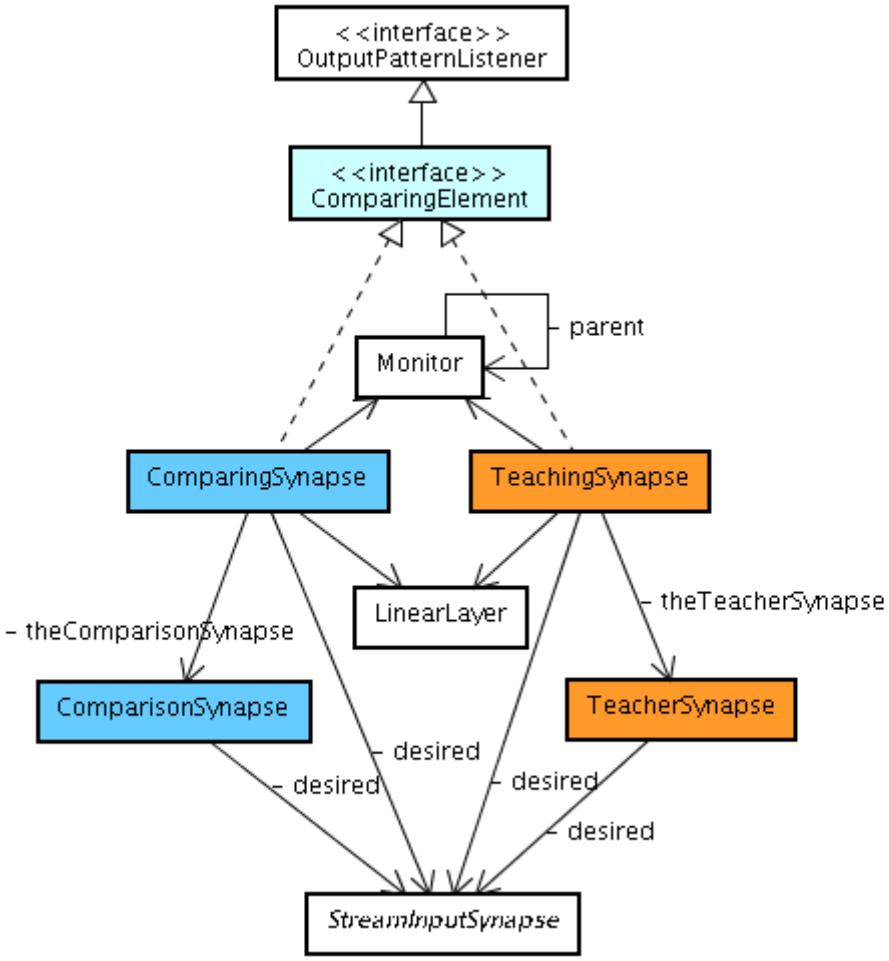
As you can see, you can add whatever learning modes you want, after that you can choose the current one simply by setting the learningMode property.

Of course you need to declare only the learner objects you want to use, not all the existing ones! And if you need to use only the basic on-line mode, then you don't need to do anything, as that learning mode is the default learner, and it's activated whenever no learners have been declared.

5.4 Technical details

5.4.1 The learning components object model

All the learning components are in the `org.joone.engine.learning` package, and its object model is represented in the following figure:



As you can see, all the above described components are represented.
The TeachingSynapse is a compound object containing, other than a TeacherSynapse object, also a LinearLayer.
When you put a TeachingSynapse within a neural network, you must simply connect it to the last Layer of the net (using Layer.addOutputSynapse) and set the *desired* property to the StreamInputSynapse object containing the desired output patterns.

Nothing else, as the TeachingSynapse will provide you with all the services needed to calculate the error to feed the neural network during the training supervised phase.

The error is transmitted, by the LinearLayer, to the attached OutputPatternListener object.

The ComparingSynspase is also contained in this class diagram, and inherits the same interface of the TeachingSynapse class (the ComparingElement interface), hence it can be used in the same manner, permitting, in this case, to compose the two different input data sets – the output and the desired one – to compare them.

As you can see, both the two families of components – Theaching/Teacher and Comparing/Comparison – belong to the same class of components, and have the same internal composition.
Both they read two external sources of data:

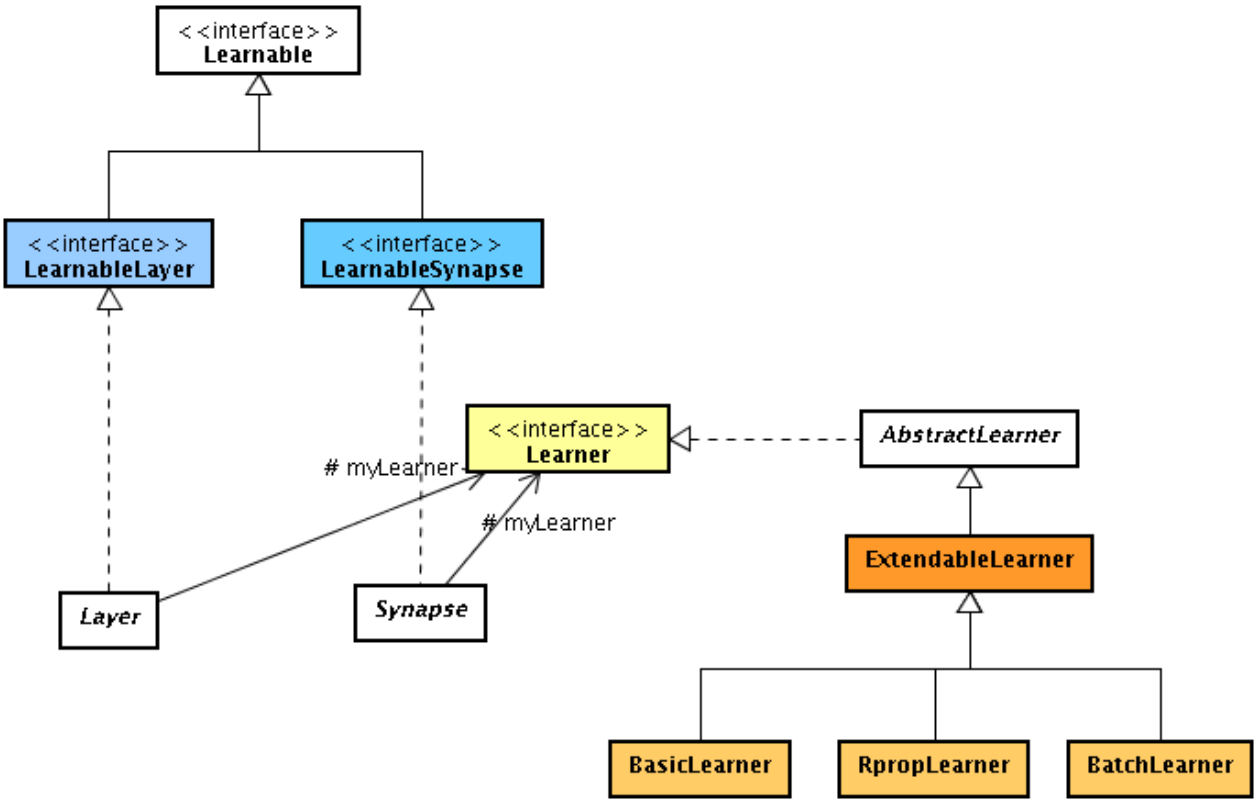
1. the **output pattern** from the output layer of the neural network and
2. the **desired pattern** from an external data source

Therefore the unique difference is represented by the pattern calculated as output:

1. The Teaching family calculates the difference between the two patterns (i.e. a scalar value representing the current training error of the neural network)
2. The Comparing family, instead, calculates the composite pattern obtained by combining the above two patterns (i.e. a vector containing the concatenation of the two patterns)

5.4.2 The Learners object model

The following is the scheme of the Learner/Learnable mechanism:



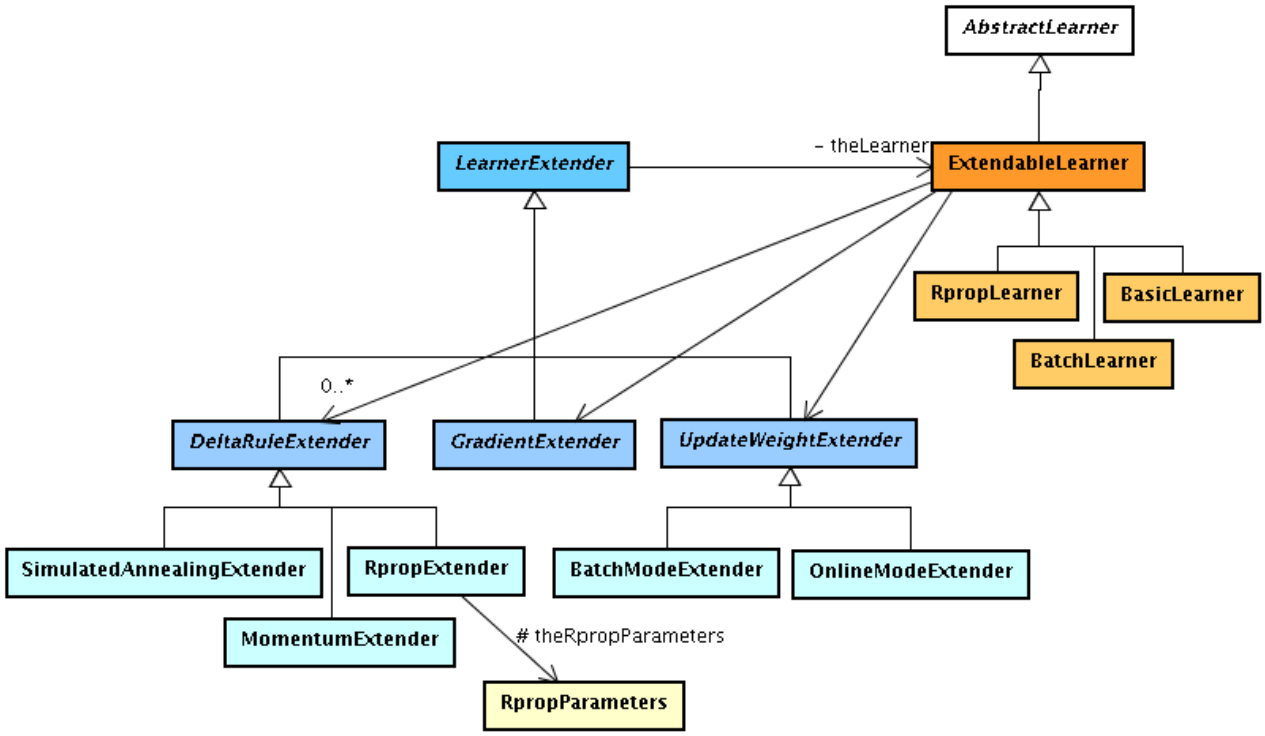
The `org.joone.engine.Learner` interface describes all the methods that each learner must implement. A Learner contains all the formulas that implement the corresponding learning algorithm, and, within each of them exist the implementations for both the Layer biases' changes (`requestBiasUpdate`) and the Synapse weights' changes (`requestWeightUpdate`).

Based on the content of the learningMode property of the Monitor, at the start of the neural network both the Layers and the Synapses receive a pointer to the active learner (represented by the 'myLearner' variable in the above diagram), so each component will be able to call the needed Learner's method according to its nature, in order to permit their biases/weights to be adjusted during the training phase.

Each component that can be manipulated by a Learner must implement the `org.joone.engine.Leaneable` interface and, as described by the above diagram, two Learnable objects exist: `LeanableLayer` – implemented by the Layer – and `LearnableSynapse` – implemented by the Synapse object.

5.4.3 The Extensible Learning Mechanism

Since the version 1.2 of the core engine (thanks to the great work made by Boris Jansen), the learners mechanism has been extended in order to permit to easily add whatever else learning algorithm, simply by extending the `LearnerExtender` abstract class, as depicted in the following figure:



The framework is based on so-called extenders, which implement a certain part of a learning algorithm. For example, certain extenders calculate the update value for the weights (e.g. standard back-propagation, RPROP, etc) other extenders implement the weight storage mechanism (e.g. online mode, batch mode, etc). By combining and creating extenders users can develop their own learning algorithms. Still, if a certain learning algorithm cannot be implemented by using the framework, for whatsoever reason, the user is still able to extend the `Learner` or `AbstractLearner` interface/class directly and build the learning algorithm from scratch.

However, one of the advantages of the learning framework is that user can use existing extenders to construct their learning algorithm and only focus on that part of the learning algorithm that differs from the functionality provided by the extenders.
For example, if a user wants to implement a new learning algorithm that uses a different delta weight update rule, the user only has to implement a `DeltaRuleExtender`. By combining the new extender with an `OnlineModeExtender` the learning algorithm becomes an online training algorithm. By combining the new extender with a `BatchModeLearner`, the learning algorithm becomes a batch mode (offline) learning algorithm.

Yet another advantage is that not only certain basic functionality can be overwritten by new extenders, it can also be combined. For example, a certain new `DeltaRuleExtender` can be combined with the `MomentumExtender` to provide the new learning algorithm with the momentum mechanism, or it can be combined with the `SimulatedAnnealingExtender` to provide the new learning algorithm with the simulated annealing mechanism.

The framework is very flexible and different techniques can be combined and easily implemented. Still the user has to verify if certain combinations of extender make sense. For example a `DeltaRuleExtender` implementing the RPROP delta weight update rule in combination with the `OnlineModeExtender` probably gives bad results, because the RPROP learning algorithm is a batch mode learning algorithm.

Let's look a little bit more in detail the learning algorithm extender framework:
The class that provides the skeleton for learning algorithms based on extenders is the `ExtendableLearner` class. This class basically implements the standard BP algorithm, however the weights are not updated. In order to update the weights (that is to store the new values) one needs to set an `UpdateWeightExtender`, currently Joone provides two `UpdateWeightExtender`'s, a `OnlineModeExtender` and a `BatchModeExtender`.

Whenever errors are back-propagated thought the network, the methods `requestWeight`(or `Bias`)`Update` are called. The first thing the method does is to call the `preWeight`(or `Bias`)`Update` method on all the extenders that are set. This way it gives any extender the opportunity to perform some action before the weights will be updated.

Next all the `DeltaRuleExtenders` that are set are executed. The back propagated gradient error is passed to the `DeltaRuleExtenders` and the can calculate the new weight update value for the current weights (or biases). The new calculated value is passed to any next `DeltaRuleExtender` if more than one `DeltaRuleExtender` is set.
This way it is possible to combine for example the standard BP together with the `MomentumExtender` and/or `SimulatedAnnealingExtender`.
After the `DeltaRuleExtenders` have calculated the delta value, the `WeightUpdateExtender` is called, which stores the values according to some storage mechanism, e.g. the `OnlineModeExtender` or `BatchModeExtender`.
Finally the `postWeight`(or `Bias`)`Update` method is called on all extenders to give them the opportunity to do some thing (clean up) after the weights are updated.

For example, to create a learner implementing the standard BP together with simulated annealing, all I have to do is create the following class:

```
public class MyLearner extends ExtendableLearner {
   public BasicLearner() {
      setUpdateWeightExtender(new OnlineModeExtender());
      addDeltaRuleExtender(new SimulatedAnnealingExtender());
   }
}
```

If we look for example at the `MomentumExtender` all it does is add a momentum to the calculated delta weight update value:

```
public double getDelta(double[] currentGradientOuts, int j, double
aPreviousDelta)
{
 if(getLearner().getUpdateWeightExtender().storeWeightsBiases()) {
    // the biases will be stored this cycle, add momentum
    aPreviousDelta +=  getLearner().getMonitor().getMomentum() *
            getLearner().getLayer().getBias().delta[j][0];
 }
 return aPreviousDelta;
}
```

We think that in this manner we have created a very powerful framework which easies the implementation of different learning algorithms, and as Joone is an Open Source project, anyone can do it.
Send us your work, and we'll be very happy to publish it!

6 The Plugin based expansibility mechanism

The core Joone engine is built to be extended and controlled by any custom class implemented by the user.
This extensibility is obtained by using plugins that can be attached to some components of the neural network.
Three main kinds of plugins exist in Joone:

1. The input plugins
2. The output plugins
3. The monitor plugins

These are described here in detail.

6.1 The Input Plugins

These plugins are very useful for implementing mechanisms to control the pre-processing of the input data for a neural network.

Several input plugins have been implemented:

- The **NormalizerPlugin** to limit the input data into a predefined range of values
- The **CenterOnZeroPlugin** to center the input values around the origin, by subtracting their average value
- The **MinMaxExtractorPlugin** to extract the turning points of a time series
- The **MovingAveragePlugin** to calculate the average values of a time series
- The **DeltaNormPlugin** to feed a network with the normalized 'delta' values of a time series
- The **ShufflerPlugin** to 'shuffle' the order of the input patterns at each epoch
- The **BinaryPlugin** to convert the input values to binary format

Other pre-processing plugins can be built simply by extending the above classes.

6.2 The Output Plugins

The output plugins are very useful to post-process the outcome of a neural network. This could be useful to rescale an output signal to obtain a range equal to that of the original input patterns.

At this moment only one output plugin exists - the `UnNormalizerOutputPlugin` class. It, as already said, serves to rescale the output values to a predefined

range, and it's useful when a NormalizerInputPlugin is used to normalize the input patterns. It can be used simply attaching it to an xxxOutputSynapse, and setting its OutDataMin and OutDataMax parameters to the desired min/max output range values.

As for each other component in the joone's core engine, obviously also in this case it's possible to build new output plugins simply extending the basic ones.

6.3 The Monitor Plugins

As mentioned in earlier, a notification mechanism has been implemented in Joone's core engine to inform all the interested objects about some events of the neural network. Using this mechanism, a plugin system has been implemented that permits useful behaviour to be added in response to events raised by the net.
This mechanism is very simple, and permits to provide the network with pre-built useful behaviours in response to particular events.
The events that can be handled are:

- the **netStarted** event
- the **netStopped** event
- the **CycleTerminated** event
- the **ErrorChanged** event

They can be subdivided into two categories:

1. **One-time** events, like the netStarted and netStopped events
2. **Cyclic** events, like the CycleTerminated and ErrorChanged events

With Joone are delivered two Monitor Plugins that permit to control some parameters of the neural network during the learning phase by handling the cyclic **ErrorChanged** event:

The **Linear Annealing** plugin changes the values of the learning rate (LR) and the momentum parameters linearly during training. The values vary from an initial value to a final value linearly, and the step is determined by the following formulas:

step = (FinalValue - InitValue) / numberOfEpochs
LR = LR – step

The **Dynamic Annealing** plugin controls the change of the learning rate based on the difference between the last two global error (E) values as follows:

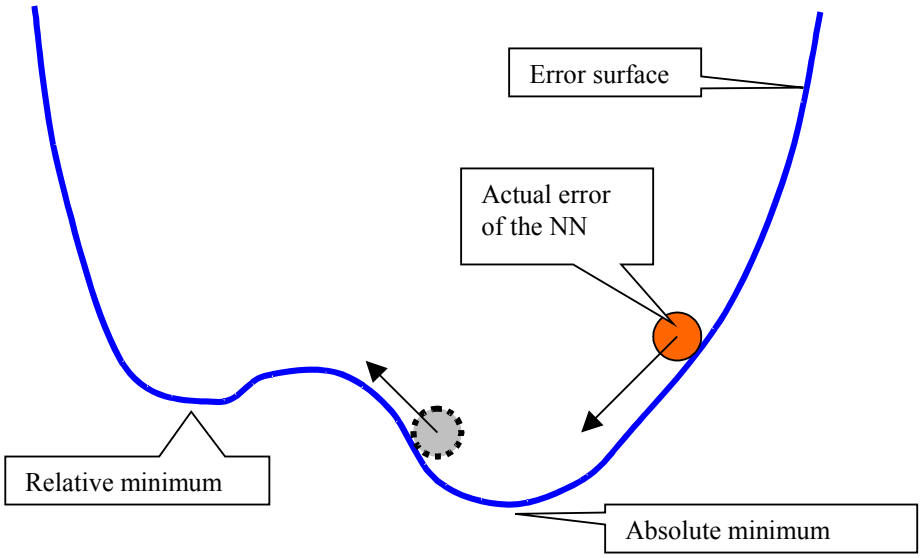
- If E(t) > E(t-1) then LR = LR * (1 - step/100%).
- If E(t) <= E(t-1) then LR remains unchanged.

The ‘rate’ parameter indicates how many epochs occur between an annealing change. These plugins are useful to implement the annealing (hardening) of a neural network, changing the learning rate during the training process.

With the Linear Annealing plugin, the LR starts with a large value, allowing the network to quickly find a good minimum, and then the LR reduces permitting the found minimum to be fine tuned toward the best value, with little the risk of escaping from a good minimum by a large LR.

The Dynamic Annealing plugin is an enhancement to the Linear concept, reducing the LR only as required, when the global error of the neural net augments are larger (worse) than the previous step’s error. This may at first appear counter-intuitive, but it allows a good minimum to be found quickly and then helps to prevent its loss.

To explain why the learning rate has to diminish as the error increases, look at the figure below:



All the weights of a network represent an error surface of n-dimensions (for simplicity, in the figure there are only two dimensions). Training a network means to modify the connection weights so as to find the best group of values that give the minimum error for certain input patterns.

In the above figure, the red ball represents the actual error. It ‘runs’ on the error surface during the training process, approaching the minimum error. Its speed is proportionate to the value of the learning rate, so if it is too high, the ball can overstep the absolute minimum and become trapped in a relative minimum.

To avoid this side effect, the speed (learning rate) of the ball needs to be reduced as the error becomes worse (see the grey ball).

6.4 The Scripting Mechanism

Joone has its own scripting mechanism based on the BeanShell (http://www.beanshell.org) scripting engine.
It takes advantage of the possibility of intercepting all the events raised by a neural network from within a Monitor plugin. To make possible the management of the neural network's events by an external script, a complete system has been implemented with the following features:

1. It is expansible, as makes possible the addition of new scripting interpreters simply by creating new classes inheriting a basic interface, without having to change any other class
2. The entire mechanism, being isolated by the rest of the core engine, does not depends on the BeanShell's libraries, making possible the distribution of a neural network without having to also distribute the scripting interpreter if the neural network does not use this feature.
3. It permits to write macros in response to any of the events raised by a neural network, permitting to implement whatever behaviour at run-time without the necessity to write and compile java code.
4. The macros are embedded in the neural network, and therefore they are stored/transported along with the neural network at which belong. This is a powerful mechanism capable to transport and remotely run some kind of 'custom logic' to control the run-time behaviour of a neural network.

The scripting mechanism contains two types of macros: **event-driven** and **user-driven** macros.

Event-driven macros are all macros associated with the defined events of the neural network. It is possible to execute these scripts in response to a net event. It is impossible to add, remove or rename these macro because they are inherently connected to the events that a neural network can raise. The user can only set their text. If no action is required for an event, the corresponding text must be cleared (i.e. set to an empty string).

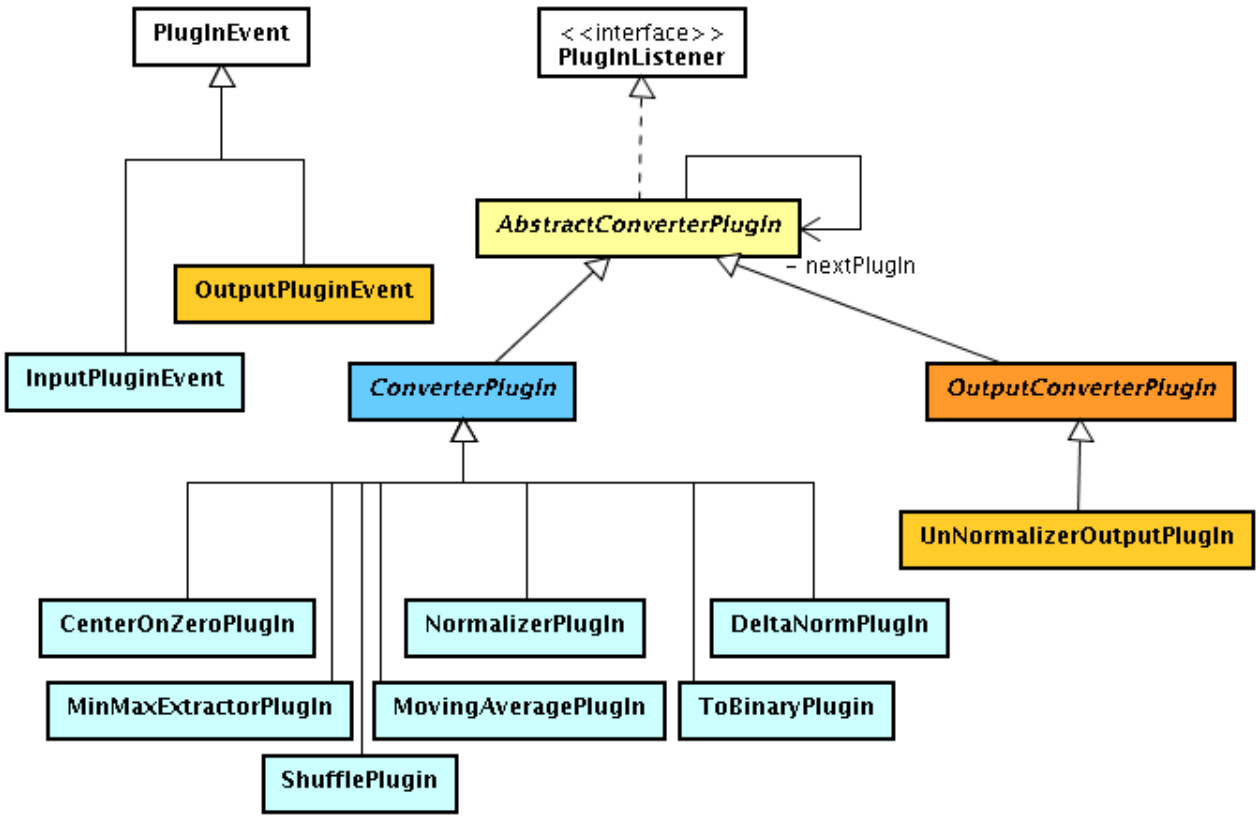
User-driven macros are macros added by the user that are executed at the user's request by calling a method. These macros can be added, removed or renamed as they are not linked to any net's event.

6.5 Technical details

6.5.1 The Input/Output Plugins object model

The mechanism, contained in the `org.joone.util` package, is based on the abstract classes `ConverterPlugin` and `OutputConverterPlugin`.

The following figure depicts the object model of the input/output plugin mechanism:



Any `ConverterPlugin` can be attached to a `StreamInputSynapse` with the `setPlugIn()` method, and can be extended to implement any required pre-processing of the input patterns read by the parent Input Synapse.
To provide the processing, classes inheriting the `ConverterPlugin` must implement the abstract method *convert()* with the necessary code to pre-process the data.

The code for the `NormalizerPlugin` is shown as an example. This class normalizes the input pattern to a range delimited by the min and the max parameters:

```
    protected void convert(int serie) {
```

```
        int s = getInputVector().size();
        int i;
        double v, d;
        double vMax = getValuePoint(0, serie);
        double vMin = vMax;
        Pattern currPE;
        /* Calculates the max and the min values of the input patterns */
        for (i = 0; i < s; ++i) {
            v = getValuePoint(i, serie);
            if (v > vMax)
                vMax = v;
            else
                if (v < vMin)
                    vMin = v;
        }

        d = vMax - vMin;
        /* Calculates the new normalized values */
        for (i = 0; i < s; ++i) {
            if (d != 0.0) {
                v = getValuePoint(i, serie);
                v = (v - vMin) / d;
                v = v * (getMax() - getMin()) + getMin();
            }
            else
                v = getMin();
            currPE = (Pattern) getInputVector().elementAt(i);
            currPE.setValue(serie, v);
        }
    }
```

Firstly, in the first for(...) loop, the min and the max values of the input data are calculated, then in the second for(...) loop the new normalized values of the input data are calculated using the following formula:

$$norm(x) = \frac{x - min(x)}{max(x) - min(x)} \cdot (UpperLimit - LowerLimit) + LowerLimit$$

Note the methods used to read/write the input values (colored in blue in the above listing):

- **`getValuePoint(row, serie)`** is used to extract an input value
- **`Pattern.setValue(serie, value)`** instead is used to write the new calculated value

The *serie* variable represents the column affected by the pre-processing action, and it is passed as a parameter to the convert method.
Because many pre-processing calculations require all the values of the input data to be read before the data can actually be processed (as in the above example), the input plugins can be attached only to a buffered input synapse. So calling the setPlugIn method on an unbuffered synapse sets its state to 'buffered'.

To allow more than one pre-processing calculation to be applied to the input data, the input plugins can be attached in sequence, building a chain structure.

To do this, the AbstractConverterPlugin itself has a setPlugIn method, like the StreamInputSynapse class. This allows one plugin to be attached to another plugin, pre-processing the input data using as many as plugins are required.
Note the auto-association link on the AbstractConverterPlugin class in the above object model.
The chained input plugins will be invoked in the same order that they have been attached in the chain.

To allow the input synapse and the attached plugins to be informed of changes to any parameter in any plugin constituting the chain, a notification mechanism based on the `InputPluginEvent` object has been implemented.

Once an input plugin is attached to an input synapse or to another input plugin, the parent object is registered as a listener to the newly attached object. Any change made to any plugin attached to the chain raises an event to its parent, which is propagated up to the chain until it reaches the parent input synapse. Here, a new pre-processing action is invoked calling the convert() method on each attached input plugin. Thus the new pre-processed input data can be calculated.

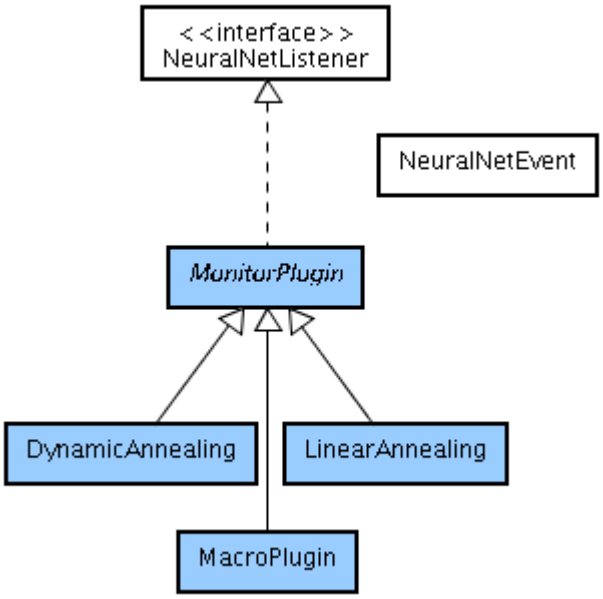
Note: For this reason, when a new input plugin is implemented, the `fireDataChanged` method **must** be called from within the *setXXX( )* method of any parameter that affects the pre-processing calculations.

As an example, consider the `setMin`() method of the `NormalizerPlugin` class:

```
/**
 * Sets the min value of the normalization range
 */
public void setMin(double newMin) {
    min = newMin;
    super.fireDataChanged();
}
```

6.5.2 The Monitor Plugin object model

The following figure illustrates the object model of the monitor plugin system contained in the org.joone.util package:



As depicted in the above class diagram, the system is based around the MonitorPlugin object, which implements the NeuralNetListener interface.
To attach a plugin to a Monitor object, the `addNeuralNetListener` method must be invoked, passing the object inheriting the MonitorPlugin as parameter.

To build a new plugin, the MonitorPlugin object must be extended. To implement the actions needed for each raised event, the corresponding `manage`*`XXX`* abstract method must be coded, where *`XXX`* is:

- `Start` to manage the **netStarted** event
- `Stop` to manage the **netStopped** event
- `Cycle` to manage the **CycleTerminated** event
- `Error` to manage the **ErrorChanged** event

The monitor plugins are very useful for dynamically controlling the parameters and/or the behaviour of a neural network.

For instance, to stop the training of a neural network when its RMSE is less than 0.01, an object could be written that extends the `MonitorPlugin` class containing the following code:

```
Public class stopCondition extends MonitorPlugin {
      protected void manageError(Monitor mon) {
            double rmse = mon.getGlobalError();
            if (rmse < 0.01)
                  nnet.stop();
      }
}
```

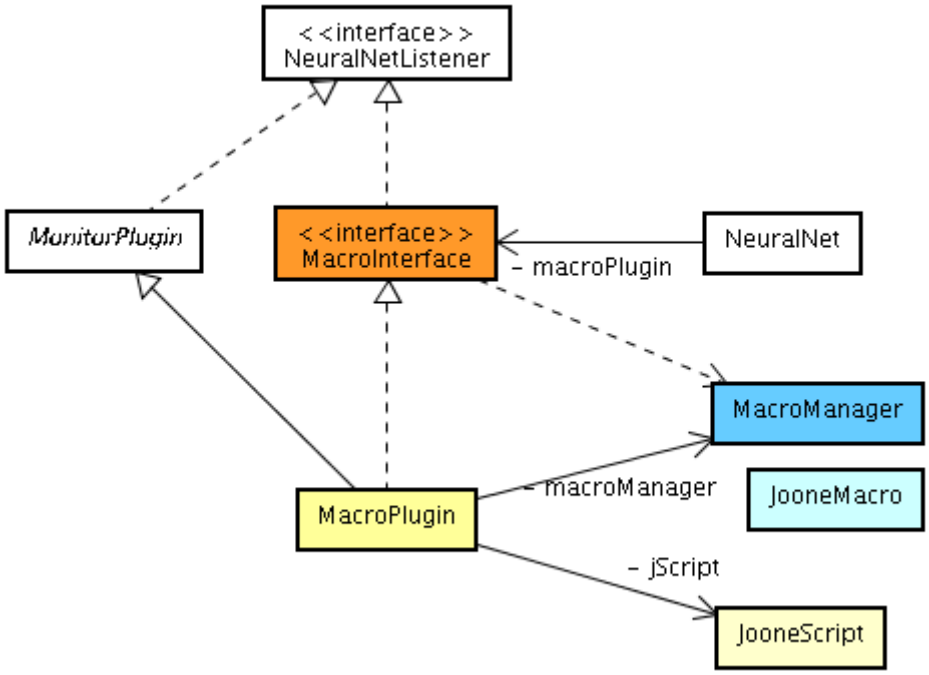
The `MonitorPlugin.rate` parameter allows the interval (number of cycles) between two events' calls to be set. This is useful for the recurring events (the *cycleTerminated* and the *errorChanged* events) to avoid calling that event

handler too often, which would reduce valuable CPU resource available to the running of the neural network.

6.5.3 The Scripting mechanism object model

The actual implementation of the scripting mechanism is based on the BeanShell scripting library, but indeed it has been built to be used with whatever else scripting library, simply by extending some basic interfaces. As far Joone 2.0, also the Groovy language is supported.

The complete object model, contained in the `org.joone.script` package, is depicted in the following class diagram:



The NeuralNet object has a pointer to the `MacroInterface` interface, which is implemented by the `MacroPlugin` object. This interface has been introduced to avoid having direct dependencies between the NeuralNet class and the BeanShell's libraries.
There are two reasons for this:

- The MacroInterface makes the addition of new scripting interpreters possible simply by creating new classes inheriting that interface, without having to change any other class, as the MacroPlugin is the unique class that in this object model must reference.
- The NeuralNet object, pointing to an interface, does not depends on the BeanShell's libraries, making possible the distribution of a neural network without having to also distribute the scripting interpreter if the neural network doesn't need to use this feature.

The `MacroManager` object is a class 'container' of all the macros defined in the neural network. Each macro is represented by an instance of the `JooneMacro` class, which contains the script's text that will be interpreted by the scripting engine when the corresponding macro will be executed.
The MacroManager contains both the two defined types of macros: **event-driven** macros and **user-driven** macros.

The following rules are applied:

- All the macro added with the `addMacro` method are inserted as **user-driven** macro
- Trying to remove or rename an **event-driven** macro results in a null action, and in this case the corresponding method returns false
- Macros can be updated by passing the new text for an existing macro as a parameter of the `addMacro` method. This saves having to remove and then add that macro.

The `MacroManager.isEventMacro(name)` returns **true** if the string passed as parameter is the name of an event-driven macro.

7 Using the Neural Network as a Whole

As we have seen, a neural network is composed by several components linked together to form a particular architecture suitable to resolve a given problem.
In some circumstances, however, it's not convenient to handle the network as a group of single components when, for instance, we need to store, reload or transport it.

To elegantly resolve these needs, we have built an object that can contain a neural network, and in the meantime also it provides the developers with a set of useful features. This object is the NeuralNet object, and resides in the org.joone.net package.

Important note: although the NeuralNet object has been initially conceived as an helper class to resolve the above needs, starting from Joone 2.0 the NeuralNet is became a fundamental class of the core engine, representing so a mandatory object to use in order to have at disposal all the features of the new engine (like for instance the single-thread mode).

7.1 The NeuralNet object

The NeuralNet object represents a container of a neural network, giving the developer the possibility of managing a neural network as a whole.
With this component a neural network can be saved and restored using a unique write and read operation, without be concerned about its internal composition.
Also by using a NeuralNet object, we can easily transport a neural network on remote machines and run it there by writing a small generalized Java program.

The NeuralNet provides the following services:

A neural network 'container'

The main purpose of the NeuralNet object is represented by the possibility to contain a whole neural network. It exposes several methods useful to add, remove and get the layers constituting the contained neural network.

The NeuralNet object, in fact, provides the user with some useful features to manage feed forward neural networks by exposing methods to add/remove layers (**addLayer(layer)** and **removeLayer(layer)**), to get a Layer by its name (**getLayer(*name*)**) and to extract the first and the last tier of a neural network (**getInputLayer()** and **getOutputLayer()**), giving either the declared input/output layers, or searching them following these simple rules:

1. A layer is an **input layer** if:
    a. It has been added by using the NeuralNet.addLayer(layer, INPUT_LAYER) method, or...
    b. It has not input synapses connected, or...
    c. It has an input synapse belonging to the StreamInputSynapse or the InputSwitchSynapse classes

2. A layer is an **output layer** if:
    a. It has been added by using the NeuralNet.addLayer(layer, OUTPUT_LAYER) method, or...
    b. It has not output synapses connected, or...
    c. It has an output synapse belonging to the StreamOutputSynapse or the OutputSwitchSynapse or the TeacherSynapse or TheachingSynapse classes

The knowledge of all these methods is very important to manage the input/output of a neural network, when, for instance, we want to dynamically change the connected I/O devices.

A neural network 'helper'

The NeuralNet object provides the contained neural network with some components useful to its work. Starting from the assumption that to build a neural network with Joone we must connect to it both a Monitor and a TeachingSynapse object (see the above chapters), the NeuralNet already contains internally these two objects.
The NeuralNet creates an instance of the Monitor object and connects it automatically to any layer added to it.
It also holds a pointer to a TeachingSynapse object and permits this to be externally set by calling the **get/setTeacher** methods.

A neural network 'manager'

The NeuralNet object is also the 'manager' of all the behaviour of the contained neural network exposing methods like **addNoise**, **Randomize**, **resetInput**, etc. taking care to apply these methods to all its contained components.

Moreover, starting from Joone 2.0, the NeuralNet object exposes the methods needed to start/stop and restart a neural network (**go**, **stop**, and **restore** respectively). It manages transparently all the code needed to run a network, both in multi and single thread modes. It also permits to run a network both in asynch and synch mode, the last one very useful in order to wait for the termination of all the network's running threads. It's very important to use it when we need to wait for the termination of the running of a neural network without the necessity to use CPU-consuming loops to interrogate the state of the network.

7.2 The NestedNeuralLayer object

This object is the fundamental component to use when we want to build a modular neural network, i.e. a neural network composed by several other neural networks.
This feature is very useful when, for example, we want to use a neural network as a pre-processing layer of another one, or we want to teach separately several network to recognize particular aspects of the problem to solve.

The NestedNeuralLayer class, contained in the org.joone.net package, comes in our aid by providing a 'container' able to hold another entire neural network. It exposes a method – setNeuralNet(nnet) – that permits to set the embedded neural network by indicating as parameter the name of a file containing the serialized form of a NN (obtained, for instance by exporting a NN from the GUI Editor).

The NestedNeuralLayer class has a property named 'learning', used to determine if the embedded neural network's weights and biases must be changed during the training phase when inserted in the main neural network.
When the 'learning' property is false, the weights of the embedded NN are not adjusted during the main neural network's training, and also the embedded NN ignores the 'randomize' and 'addNoise' commands given to the main neural network, in order to preserve the weights learned in the initial phase.

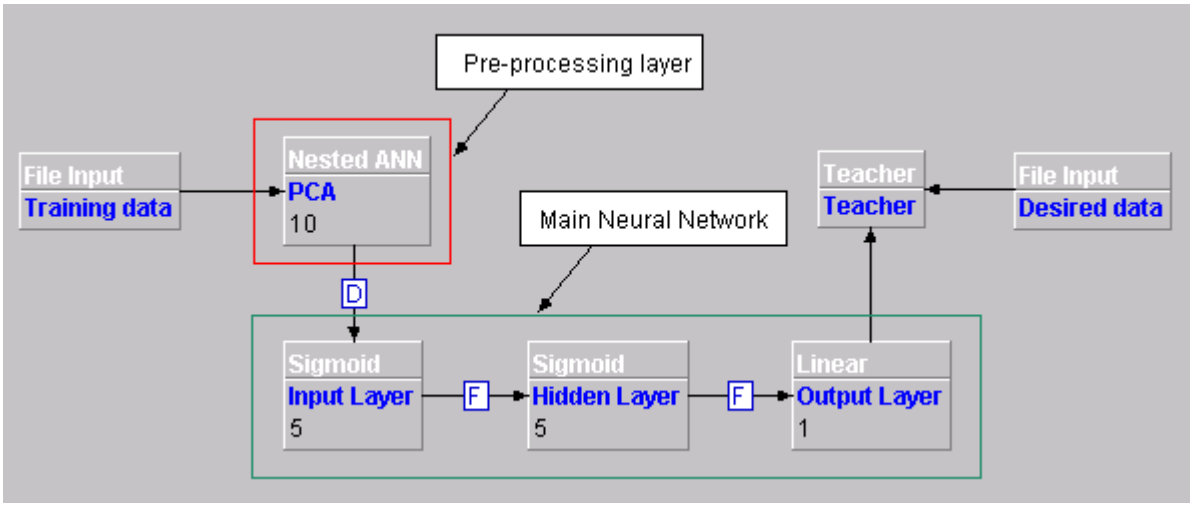
The purpose of this property becomes clear when we explain how the NestedNeuralLayer is normally used.

Let us want to use a PCA as pre-processing layer of a neural network, and we want to train the same NN until we find a good one having a low RMSE. In this case we need to execute the following steps:

1. Build a PCA NN (by using the SangerSynapse) and train it in unsupervised mode
2. Export it to a file in a serialized format (after having removed the i/o components used during the training)
3. Build the main neural network, and insert a NestedNeuralLeyer as first layers
4. Import the above serialized PCA NN into the NestedNeuralLayer
5. Set to false the 'learning' property of the NestedNeuralLayer (it should already be set to that value by default)
6. Set to true the learning mode of the main NN
7. Randomize the weights of the main neural network
8. Start the training phase
9. Repeat the steps 7 and 8 until you get a good RMSE

As you can see, at the steps 7 and 8 we shuffle the weights and then train the main neural network several times, but we don't need to do so also for the embedded one, because we have already trained it at the step 1, hence at the step 5 we need to freeze the learned weights of the embedded NN.

The following figure depicts the final neural network as it would appear in the GUI Editor.

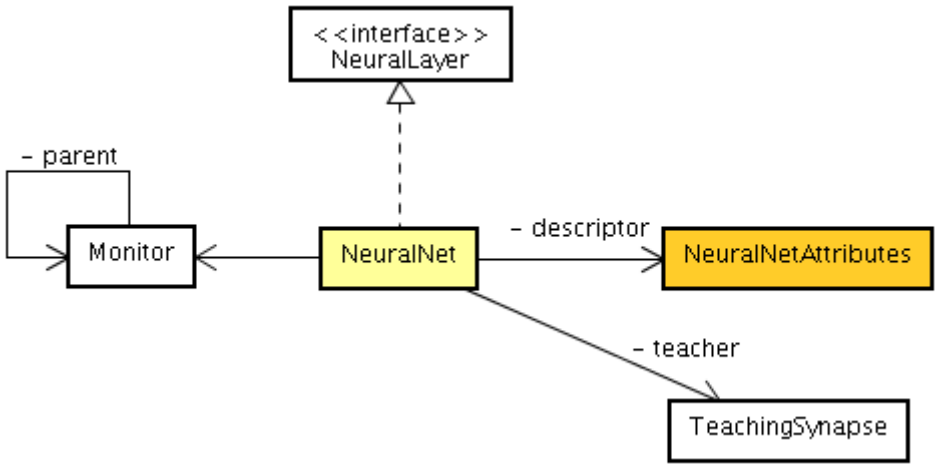


In this example the PCA is used to reduce the input layer's size from 10 to only 5 nodes, reducing in this manner the neural network's complexity.

In fact we need only to train 30 weights (5x5+5) instead of 55 (10x5+5), reducing in this manner the training time and limiting the curse of dimensionality.

7.3 Technical details

The following figure depicts the object model of the org.joone.net package, showing the NeuralNet and its link with other classes and interfaces of the core engine:



First of all, we must note that the NeuralNet object implements the NeuralLayer interface – the same implemented by the Layer object – making possible to use it as whatever else Layer in a neural network; in this manner it's possible to build very complex neural networks where each Layer could be represented itself by an entire neural network.

As you can see, the NeuralNet object contains a pointer to an embedded TeachingSynapse and a Monitor object, providing, in this manner, the objects necessary to buid correctly a neural network.

The NeuralNet class, also, contains a pointer to the NeuralNetAttributes class. It contains several parameters of the attached neural network useful during the training or validation phases (like, for instance, the neural network's name and the last training and validation errors).
This class, of course, can be extended adding new custom attributes.

Anyway if we need to add dynamically new attributes to a neural network without being constrained to write and compile new java classes, the NeuralNet object contains a mechanism to store custom parameters at run time, based on an Hashtable that stores key-value pairs.

They can be used calling the following methods:

- `void setParam(String key, Object value)` – to store a key-value pair

- `Object getParam(String key)` – to retrieve a saved parameter given its key

This possibility is very useful, for example, when the neural network is trained remotely in a distributed environment, because some parameters can be set during the remote training phase and then recalled and used by the central machine where the results must be collected.

This technique is made even more useful by the possibility to set/get these parameters from within the java scripting code.
A good example of the use of this mechanism is shown in the MultipleValidation sample provided with the core engine distribution package.

8 Common Architectures

8.1 Modular Neural Networks

As said in the previous chapters, Joone exposes a modular engine that permits to build any neural network architecture, and also it permits to build modular networks, i.e. neural networks composed by several other embedded neural networks.
The central component of this feature is represented by the NestedNeuralLayer. In the following paragraph we'll illustrate a classical example by building a modular neural network to resolve the parity problem.

8.1.1 The Parity Problem

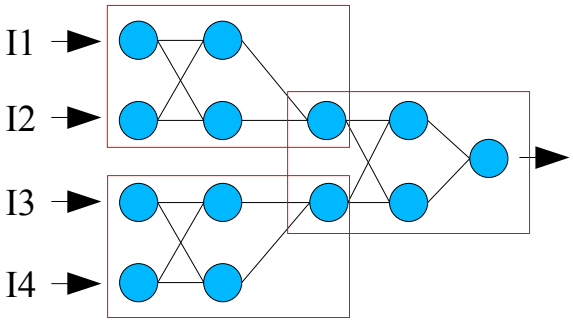
This is a classical problem, like the XOR, used to show the learning capabilities of the neural networks applied to non-linearly separable problems.

The truth table of the 4-bits parity problem is the following:

I1	I2	I3	I4	Output
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

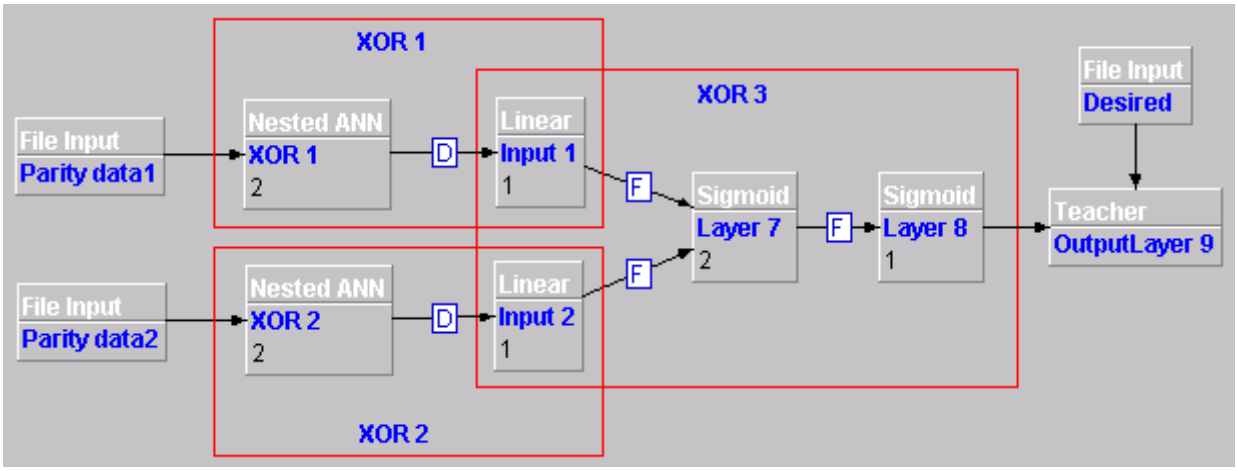
The 4-bit parity problem can be resolved by a feed forward neural network with one hidden layer, but if you try to do it, you'll notice that the training time is very long, and sometime the neural network will not learn to resolve the problem.

A better approach is represented by a modular network composed by three XOR NNs, as depicted in the following figure:



The three XOR neural networks are marked with a red rectangle, and you can see that the output nodes of the first two neural networks are used as input nodes of the last one.
Let us show now how to build the above architecture with joone.

1. First of all, build a XOR neural network with the GUI Editor and train it until the RMSE goes below 0.01
2. Export that neural network (by using the 'File->Export NeuralNet...” menu item), after having deleted all the i/o and the teacher components
3. Create a file named 'parity.txt' and write into it the parity truth table using semicolons as columns separator, like in the following example:
```
0;0;0;0;0
0;0;0;1;1
0;0;1;0;1
....
1;1;1;1;0
```
4. Build a neural network following the architecture shown in the figure:



You can recognize the three XOR NNs, bordered by the red boxes as in the previous figure.

Now perform the following tasks:

4. Import the above exported XOR neural network into the two NestedANN components (named 'XOR 1' and 'XOR 2' in the figure)
5. Set the two File Input components in order to read the parity.txt file, setting their advancedColumnSelector as follows: “1-2” for 'Parity Data1' and “3-4” for 'Parity Data2'. Remember also to set to false the 'stepCounter' parameter of 'Parity Data2'.
6. Set the desired File Input component in order to read the column 5 of the parity.txt file.
7. Open the ControlPanel and set:
   a) learning = true
   b) learning rate = 0.7
   c) momentum = 0.7
   d) training patterns = 16
   e) epochs = 5000
8. Run the training phase.

You should see the a descending RMSE value, that demonstrates that the neural network is able to learn the parity problem by using a modular architecture.

8.2 Temporal Feed Forward Neural Networks

In this chapter we'll show some potential application of the neural networks in the field of the time series elaboration in order to predict the future values given the past history of the temporal series.

8.2.1 Time Series Forecasting

First of all, we want to warn about the difficulties that arise when we try to make time series prediction applied to problems of the real life, like weather or financial predictions.

Reading the emails that we receive from the users of Joone, we know that about the 60% of them want to use the neural networks to make financial predictions.
Be aware: may people think that a neural network is like the Aladdin's lamp, but soon they discover that the reality is different, and that it's very difficult to obtain good results.

Don't waste time: very few people know that by using simply the past prices as training data is not enough. You must fight and eliminate your main enemy: **the noise**.

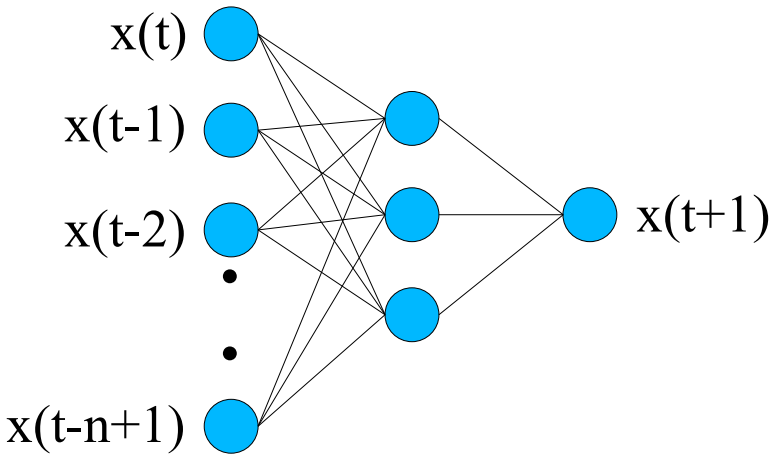
Therefore the following paragraphs want to give you just an initial knowledge about the most famous and used techniques, but you need to try many and many different architectures and pre-processing techniques in order to have some possibility to obtain some good result.

8.2.1.1 Preprocessing

To make financial forecasting is one of the most famous applications of the neural networks. In this section we want to explain the use of a particular pre-processing technique useful to make trend predictions.

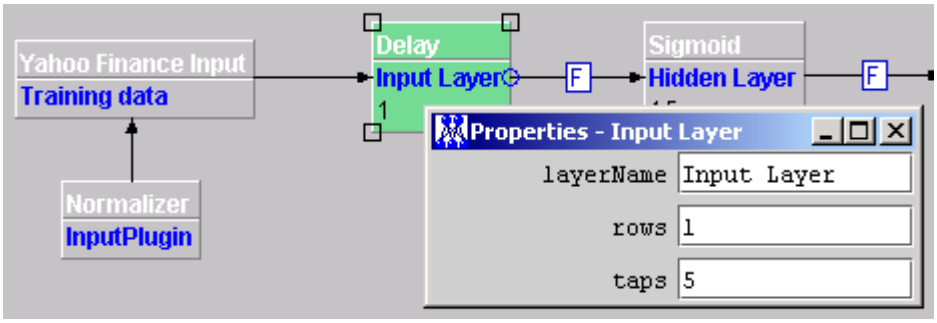
One of the most used techniques is to sample the time series at discrete moments (hourly, daily, weekly, etc.) and use the measured values as input patterns of the neural network.

Because a time series, to be predictable, needs to have an internal dependency on the past values (otherwise the time series would be just a noisily random sequence), a common pre-processing technique is represented by feeding a neural network with a temporal window of the time series, like depicted in the following figure:



As you can see, each input pattern is composed by the values at times t, t-1, t-2,...,t-n+1 where n is the temporal window's size.

How to obtain a temporal window of a given size starting from a single-column stored time series?
Joone provides the user with a component, the DelayLayer, useful to create a temporal window to feed the input layer of a neural network.
Look at the following figure containing an example built with the GUI editor of Joone:



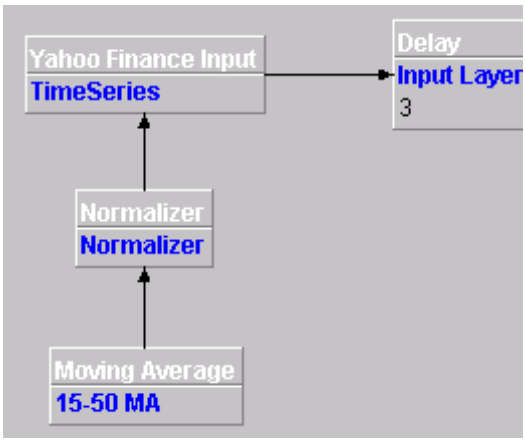
As you can see, we have used a YahooFinance input component to get the stock prices time series from Yahoo, and have connected it to a Delay layer.
The properties panel for this component, other than the rows, permits to set the 'taps' parameter, that indicates the size of the temporal window we'll use to feed our neural network.
By setting taps to '5', we obtain a window of size 6 composed by the following values:

[x(t), x(t-1), x(t-2), x(t-3), x(t-4), x(t-5)]

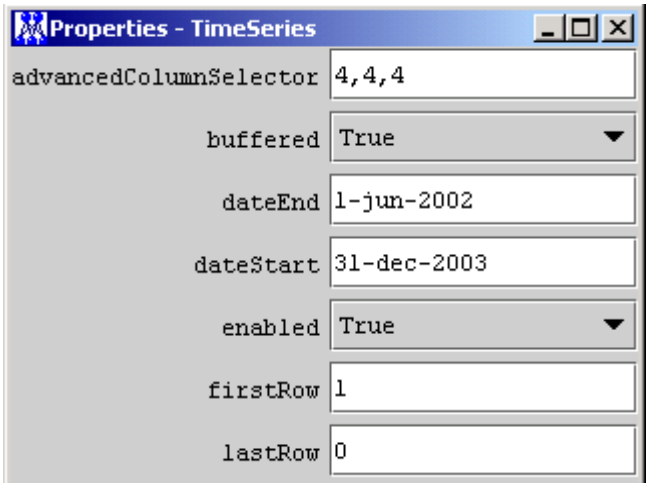
Remember that the size of the temporal window is always equal to (taps+1), because the Delay layer outputs also the actual value x(t) of the time series.

Of course this is just an example, and you can experiment several configurations, either using windows of different size, and/or using as input not

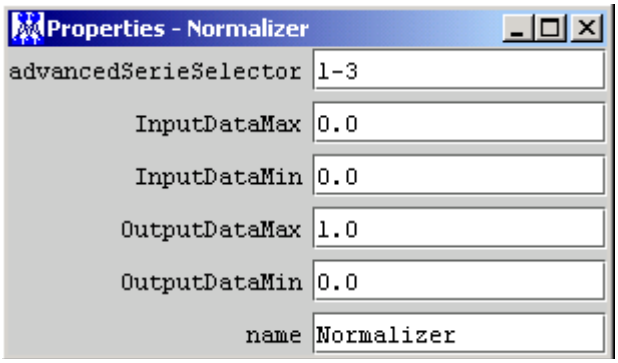
only the 'raw' data, but also some pre-elaborated values, as for instance the N-days' average (calculated by using the Moving Average Plugin component attached to the Yahoo Finance Input), as illustrated in the following figure:



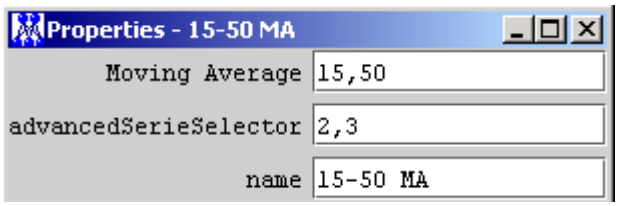
In this example we want to train the neural network using the 15 and 50 days moving averages.
The YahooFinance component has the following settings:



As you can see, the AdvancedColumnSelector has been set to “4,4,4”, so the fourth column (the Close value) will be extracted three times, and now we'll see why.
As the data must be normalized, we have used a NormalizerPlugin having the following properties:



the AdvancedSerieSelector is equal to “1-3”, thus we will normalize all the three input values between 0 and 1.
And now the MovingAverage Plugin settings:



where you can see that we calculate the 15-days and the 50-days moving average (property Moving Average set to ”15,50”) respectively on the second and third column (property AdvancedSerieSelector set to ”2,3”).

The result is that we'll feed the neural network with the following three normalized values of the time series:

1. The raw daily Close values
2. The 15-days moving average of the Close values
3. The 50-days moving average of the Close values

Now it should be clear why we have extracted three times the same value from the YahooFinance component.
Of course, in this case, the Delay Layer must have rows=3 and taps=5 (or the windows' size we have chosen to use).

That said, we now need to decide how we'll train the neural network, and to do it, we must to set the desired data of our training phase.
Normally, as desired data, the value at time t+1 is used, so the network is trained to predict the next value of the time series in base of the past n values.

Indeed it's very difficult to predict the exact value of a noisily time series, hence we want to explain a different technique to make trend predictions, instead of the next day's exact Open/Close/High/Low values prediction.

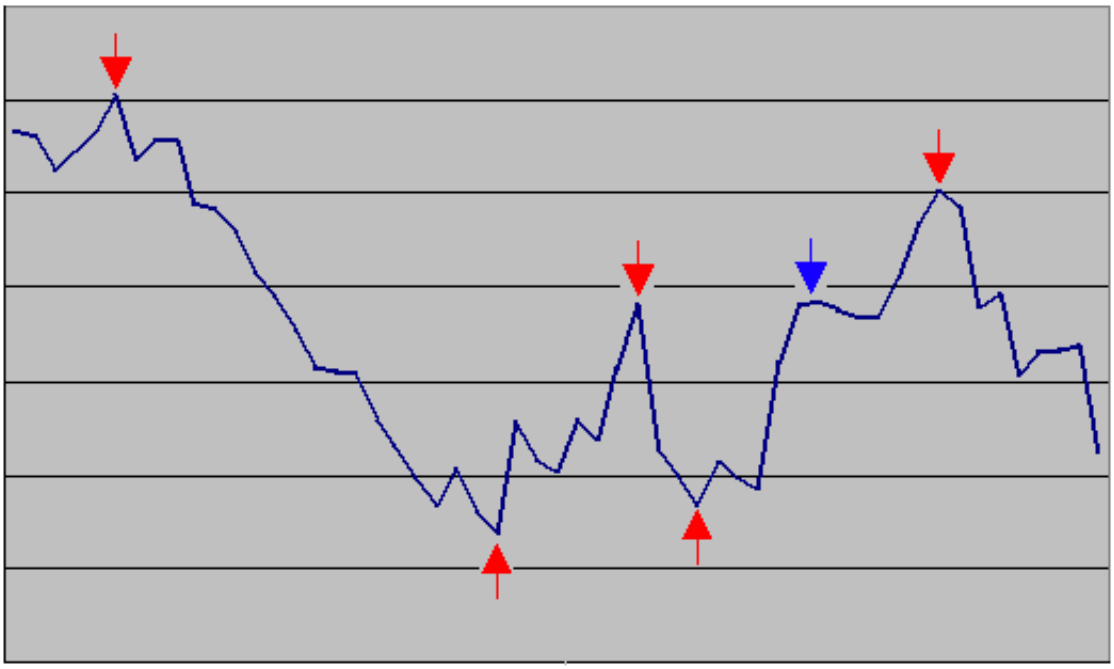
8.2.1.2 Trend prediction:

This technique tries to predict the future prices at a short-medium interval of time (from 2 to 10-15 days) using as input the past history of the prices. Using this technique, we don’t need to know the value of the next day close, but simply the future direction (up or down) of the observed market, so to take a decision about our trading position (long/short – buy/sell).

The question is: how do we teach a neural network fed with the past history of a stock? The response is very simple. Remember that this paragraph deals with the trend prediction technique, hence we don’t need to predict the exact close value of the next trading day, as we found our trading strategy on the predicted trend (up or down).

What we need to predict, in other words, are the turning points of the market we're dealing with.

Look at the following chart:



We should trade in correspondence of the red arrows to make money, buying on the lowest values and selling at the highest ones.
A good trading system should raise a signal only when a true turning point is reached, avoiding to generate false signals, as, for instance, that one indicated by the blue arrow, where the market goes down only for a little percentage before to continue to raise.
As you can see, we don't need to predict the exact values of the daily market closes, as we're interested to predict only the right turning points.

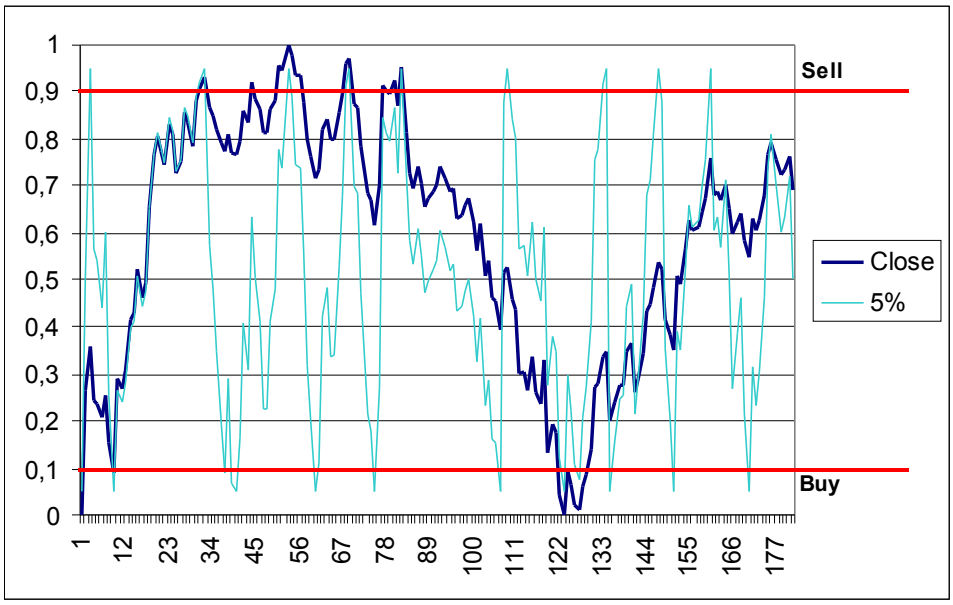
To do this, Joone owns the TurningPointExtractor plugin that calculates exactly the ideal trading signals, as explained in the above figure.
It has a 'min change percentage' property that serves to indicate what is the minimum % change between two turning points to generate the corresponding signals. It must be set to a value not too small, to avoid to generate too many signals (many of which could be false).

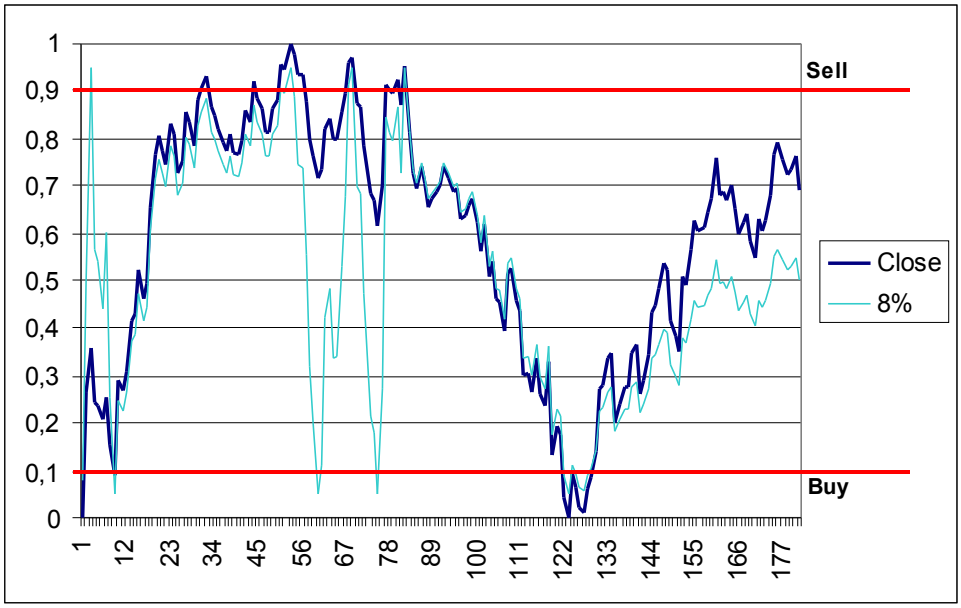
The algorithm is the following:

- When the market rises more than the desired % change, the previous lower value is flagged as a 'buy' signal, and the corresponding output value is set to 0.
- When the market declines more than the desired % change, the previous higher value is flagged as a 'sell' signal, and the corresponding output value is set to +1.
- The desired values for days between the above two points are normalized by interpolating to values within the interval 0 and +1.

The following two figures show the output of the turning point extractor plugin for a given time series.
Their min. percent change parameter is set to 5% and 8% respectively.

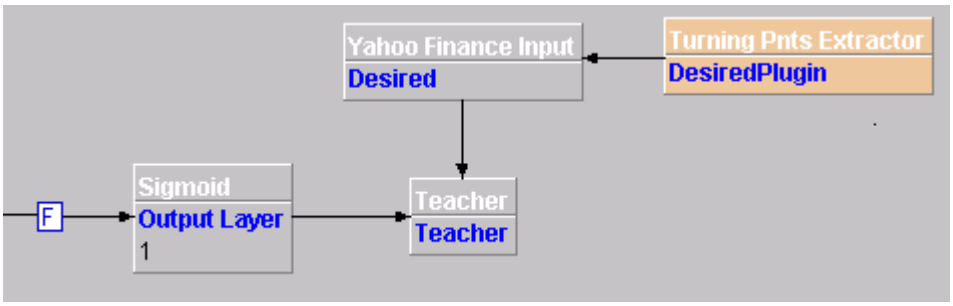
As you can see, setting the generation of buy/sell signals only when the output value is lower than 0.1 and higher than 0.9 respectively, the number of signals generated for the 5% setting is greater than those generated for the 8% case.





It doesn't exist a fixed rule to calculate the optimal percentage, and you need to try several configurations until you get good results in terms of good generalization capacity of the resulting neural network.

As we want to predict the turning points of the time series, we need to teach the network to recognize them, hence the TurningPointExtractor plugin must be connected to the desired input signal of our neural network, as depicted in the following figure:



In this manner the signals generated by the plugins will be used as the 'desired' data on which the neural network will be trained.

To summarize, we need to train a neural network teaching it to recognize the turning points of the observed market. To do this, we must feed the neural network with, for example, a temporal window of the normalized past input data, and we must use the turning point extractor to generate the desired values for the supervised learning phase.
After that, we interrogate the net giving as input the last closes normalized with the same techniques seen above and, only when the output of the net is:

- lesser than 0.1, the signal is **BUY**
- greater than 0.9, the signal is **SELL**

Of course, as already said, we must do several experiments to set both the above limits and the other parameters' right values, in order to obtain acceptable results.

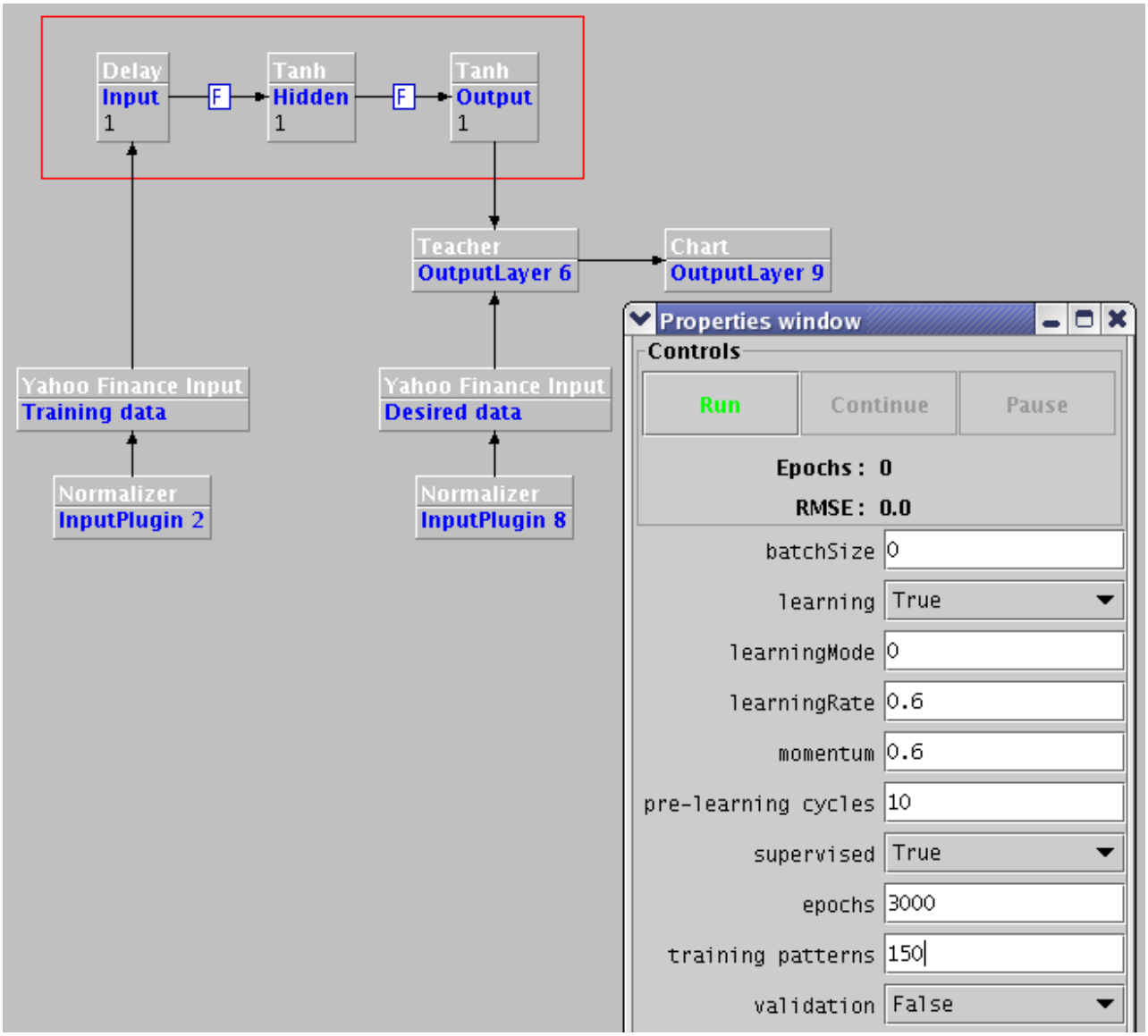
Good luck! ...and if you obtain some good result, remember to make a donation to joone :o)

8.2.1.3 Dynamic control of the training parameters

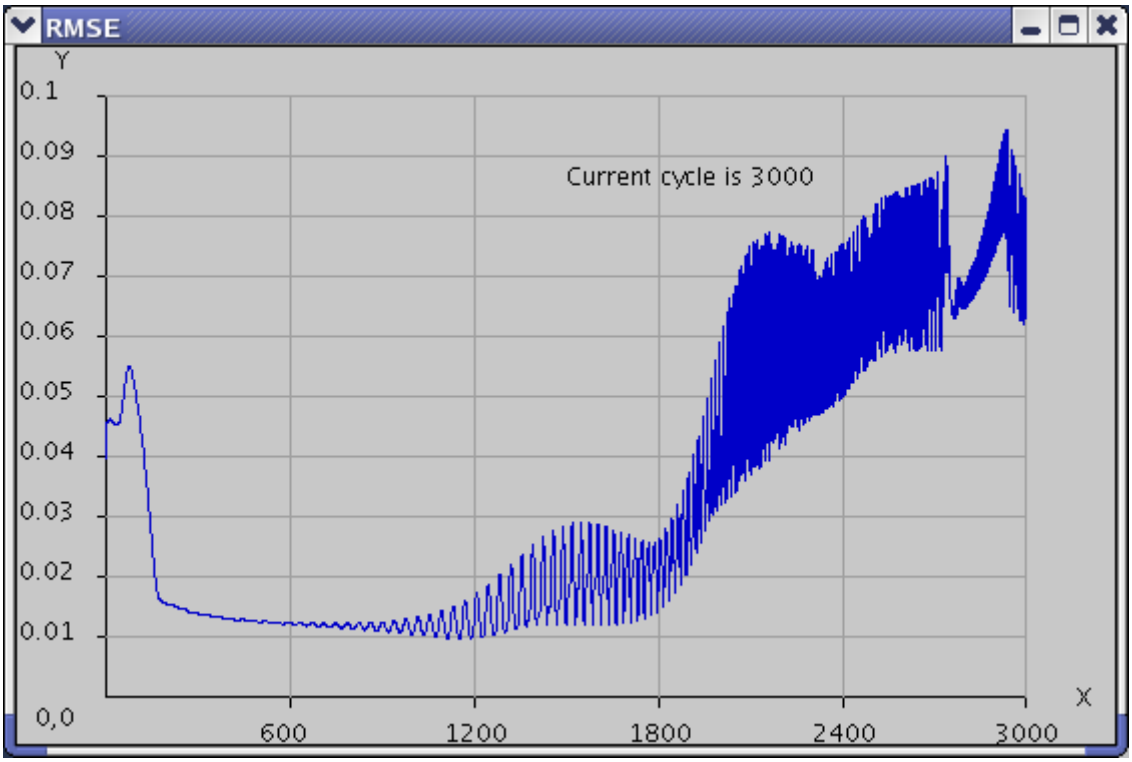
The learning rate used to train a neural network is a crucial parameter, and the choose of the right value is determining to have good results, especially for noisily time-series prediction.

As said in the paragraph 6.3, the learning rate represents the 'speed' on the error surface with which we search the optimal minimum. A value too big would cause an oscillation around the minimum, while a value too low would cause a very long training time.

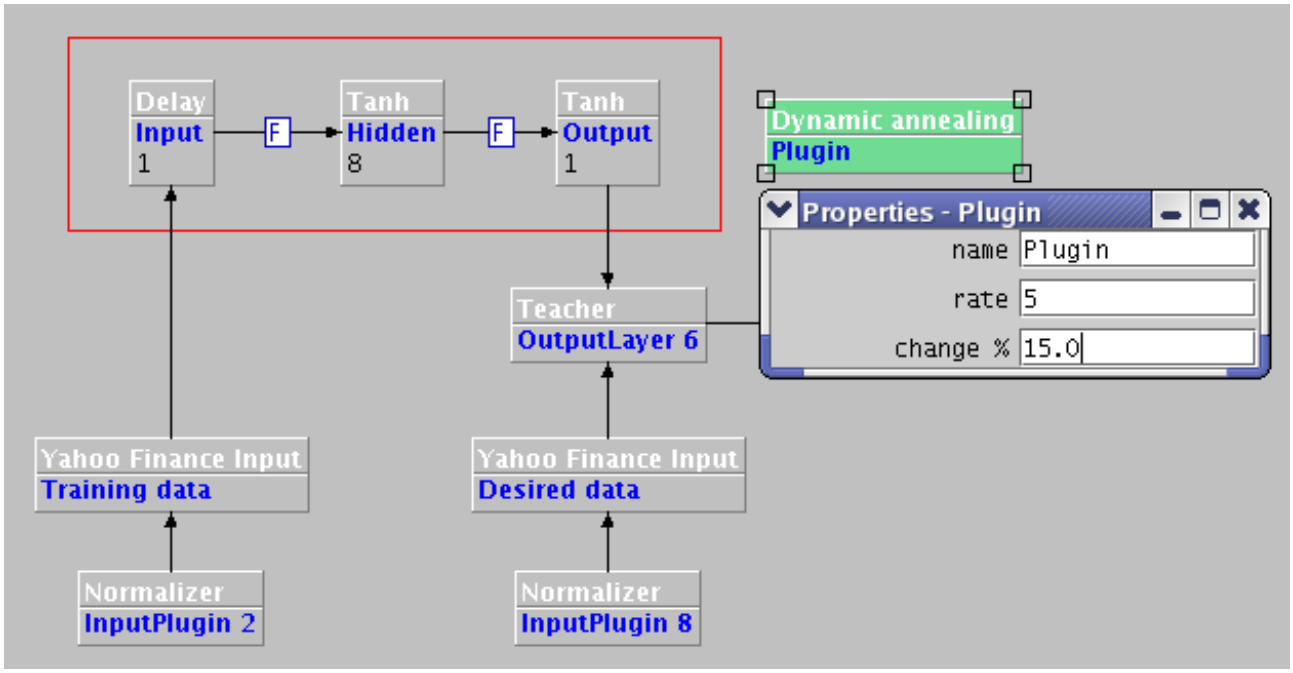
To resolve this dilemma, we can use the DynamicAnnealing plugin.
Look at the following neural network:



It's a neural network to make financial predictions (it's just an example, of course), and in the Control Panel you can see all the settings we have used (note the learning rate and the momentum both set to 0.6, a relatively high value).
When we train the above neural network we obtain a RMSE like the following:

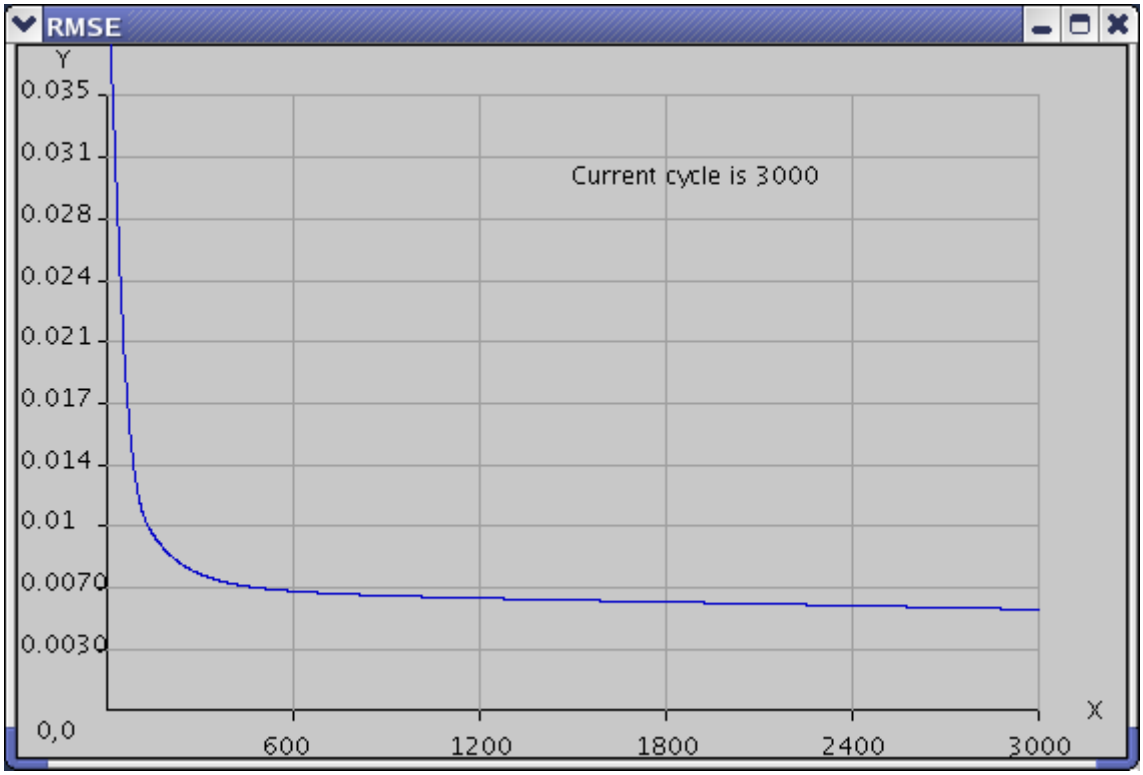


That's horrible! Due to the wrong settings, the neural network has not been able to learn the time series we have used as input, and the final RMSE is really too bad.
Now we'll insert a DynamicAnnealing plugin, as shown by the following figure:



the DynamicAnnealing's rate is set to 5 and the change to 15%. These values mean that the plugin will check the training RMSE value each 5 epochs, and when the last value is greater than the previous one, it will decrease the learning rate of 15%. Note that the Dynamic Annealing component is not attached to any component of the neural network.

Here is illustrated the resulting RMSE:



Good! We have eliminated all those horrible oscillations, and the final RMSE after 3000 epochs is very small.

Of course this is just an example, and maybe you'll obtain different results with your own neural network, but remember that by trying different values for the DynamicAnnealing's properties, you'll be always able to regularize the learning of your network in case of uncontrolled oscillations.

8.3 Unsupervised Neural Networks

8.3.1 Kohonen Self Organized Maps

This tutorial is intended to give a basic example of how to perform image / character recognition using SOM / Kohonen neural network architectures.

8.3.1.1 Example: a character recognition system

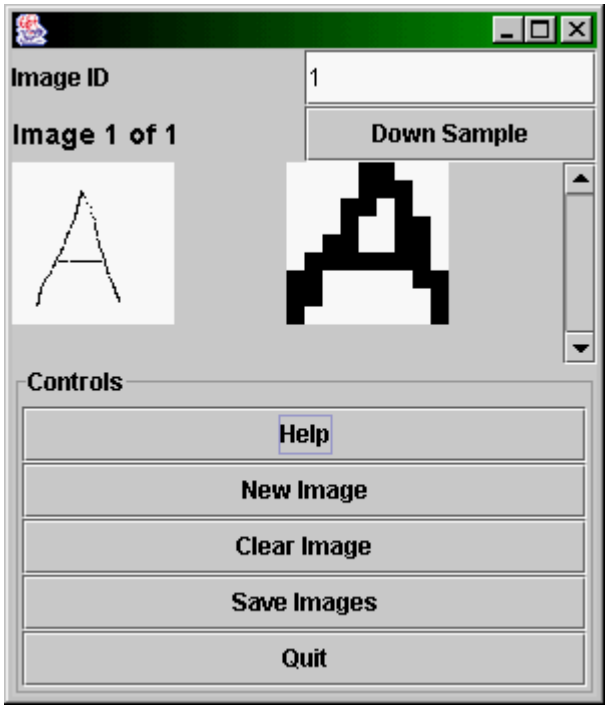
This tutorial uses a basic application called org.joone.samples.editor.SOMImageTester, and you can launch it from within the GUI Editor simply by clicking on the 'Help->Examples->SOM Image Tester' menu item.

You can use the sample application to draw basic black and white colour images and save the output into a file format that Joone recognises.

The example presented in this tutorial teaches the user how to setup a network that recognises the characters 'A' and 'B'. The reader can use this technique to setup a network that will recognise an arbitrary number of characters.

Sample Application Quick Guide

The sample application is fairly self explanatory but you can use the guide below in order to use the application.

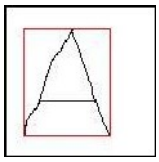


Features

Drawing Area - The high resolution 'A' image shown above is where the user can draw custom images.

Image ID - This is the identify of the image, you can use this number to mark what character the image is. Only numbers can be entered here. I.e a 1 could mean character 'A' and 2 could be 'B'.

Down Sample - This allows you to preview the down sampled image after drawing. To obtain the down sample the application first crops the image in the draw area. The image is cropped by obtaining the left most black pixel , top most black pixel etc to find the bounds of the cropped image. See the image below.



Secondly the cropped image is scaled down to a 9x9 image. The image is scaled by splitting the cropped image into a series of grids relating to each pixel in the 9x9 down sampled image, then if a grid in the cropped image contains a black pixel then the relevant pixel in the 9x9 down sampled image will contain a black pixel. The application automatically down samples each image when the user saves the the images to a file.

Help

This presents some basic help on the application.

New Image

Creates a new image for drawing a character/image into.

Clear Image

Removes all the black pixels from the current image.

Save Images

Allows all images to be saved to a Joone format file for use in a File Input Synapse. The format is 81 pixel inputs followed by the image id.

Quit

Allows you to quit the application.

Data Setup

Start the example application SOMImageTester. See the basic guide above on how to use the application.

First we need to create several 'A' character images and several 'B' character images that will be used in training and testing.

Draw the 4 'A' characters in the drawing panel clicking on New Image when you have finished each one. The down sample button can be used to see what each character looks like down sampled. When you have finished drawing the 4 'A' characters then draw four 'B' characters. Then use the Save Images button to save them out to a file, remember the file name and location we will call this 4As4Bs.txt in this example.

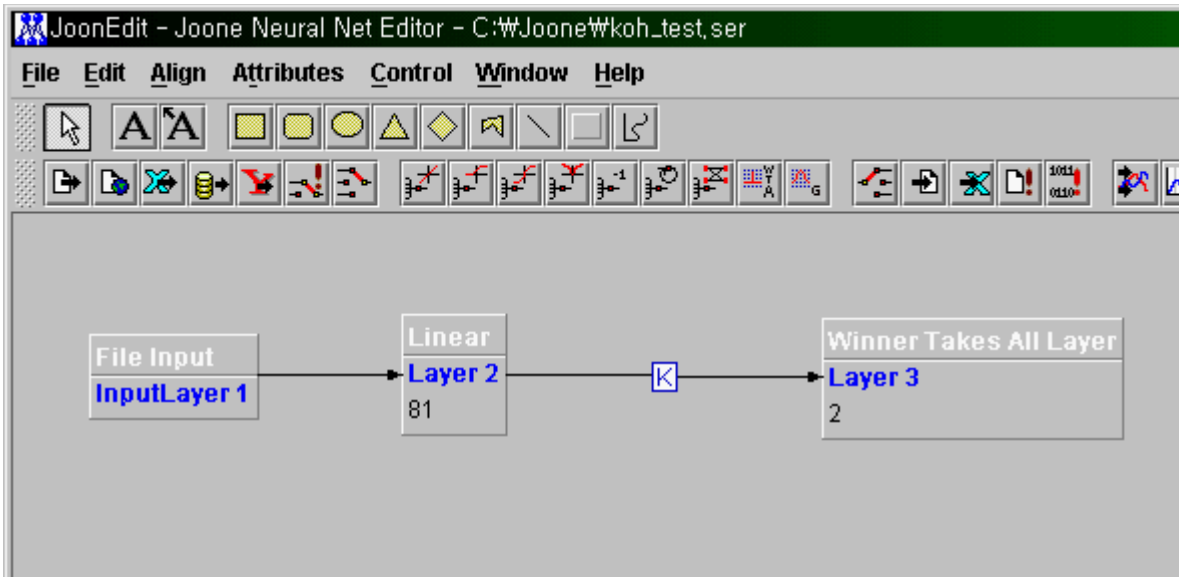
Note the more samples of a specific character you draw will mean the network is better able to recognise that character. You'll have noticed that the image gets cropped and down sampled, this is to stop the network from just recognising the character's size.

We now need a couple of test character's. Close and re-open the application , draw one 'A' character and save it we will call this testA.txt. Close the application again and re-start, this time draw a 'B' character and save the file we will call this testB.txt.

Neural Network Setup

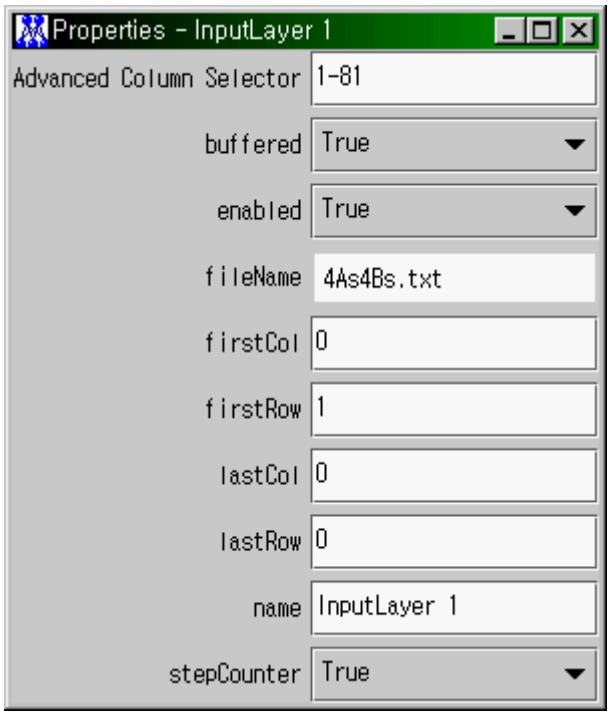
For the neural network we will be using SOM components thus the network will be unsupervised. We will need to input the previously produced file into a linear layer of 81 inputs. This will by be fed to a Winner Takes All layer via a Kohonen Synapse.
We can use a File Input Synapse to load the file.

See the image below ...



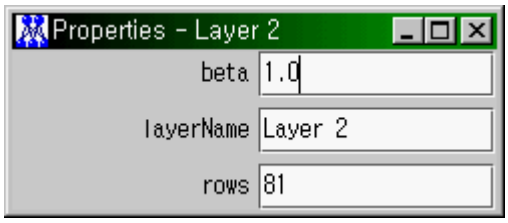
Note that the Winner Takes All layer has two neurons, this is to ensure it classifies out two characters.

Input Layer Properties



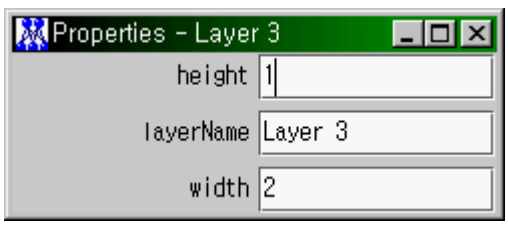
Note our input images have 81 inputs i.e the 9x9 down sampled image that the application made earlier.

Linear Layer Properties



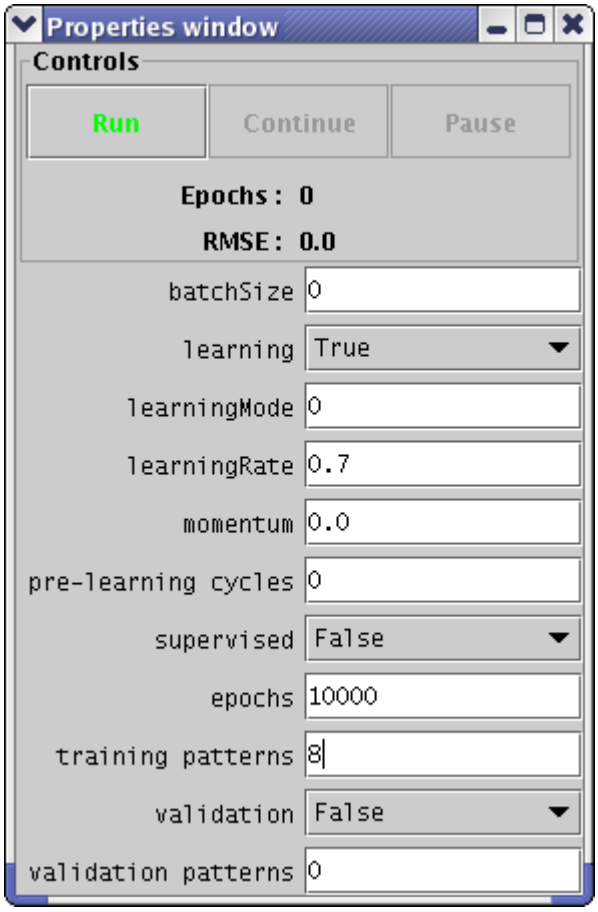
Note the rows here must match the inputs from the file input synapse.

Winner Takes All Layer Properties



Note the height or width should be 2 and 1, either can be 2 but not both. This ensure the layer contains 2 neurons for our two character classification.

Control Properties



Training The Network

Ensure the network has been set up as in the previous section. The run the network. When it has finished 10000 epochs it should have learned how to recognise the character 'A' and 'B'.

We need to find out which neuron fires on an 'A' character and which one fires on a 'B'.

We need to attach a file output synapse to the Winner Takes All Layer. Do this now and in the file output synapse set the file name to something like test.txt, in the control panel set the number of epochs to 1 and the learning property to false.

Run the network again and examine the text.txt file, you should see 8 rows and two columns. The column represents the neuron and the row the character they are trying to recognise i.e 1-8. We now that the first four characters were the character 'A' and the lest four were 'B' characters. Check that the test.txt

contains 1.0 in the same column for four rows then 1.0 in the other column for the last four rows. On hour network it came out like this ...

```
0.0;1.0
0.0;1.0
0.0;1.0
0.0;1.0
1.0;0.0
1.0;0.0
1.0;0.0
1.0;0.0
```

So we now know that by looking at the first four rows neuron 2 fires for character 'A' and neuron 1 fires for character 'B'. It could be the other way round for you.
If at this point the it is not clear i.e neuron 2 fires for both an 'A' and 'B' then you might not have setup the network correctly or it may need more training.

Testing The Network

To test the network, modify the file name in the file input synapse, select the testA.txt in order to test a character 'A'.
We have only one character in this file so in the control panel set the validation patterns to 1 and the learning mode to false. Run the network again. Examine the test.txt file, check if the correct neuron fired. In our case it was correct ..

0.0;1.0

Neuron 2 fired indicating that the network thought it was a character 'A', it is correct.

You can do the same for the testB.txt file.

Using The Network

It is possible to use this network in your own application but your custom application must present 81 inputs which are written as row1 x,x+1,x+2,x+3,...,x+9 , row2 x,x+1,x+2,x+3,....x+9 , row3 ..... , row9 x, x+1,x+2,...,x+9. Direct input from memory will require the Memory Input Synapse.
An on pixel is represented as 1.0 and off 0.0.
The network can obviously not handle colour just black (on) and white (off).
Your application will also have to crop the image and down sample it to the correct size.

Further Work

Image recognition is a fascinating field and you'll probably want to experiment in recognising different images / objects.

It should be useful to produce an Image Input Synapse that will enable users to present images from files or Java images. If (and when) this will be available, then you could use this to easily load images into the network for training and running.

Whatever contribution in this area is welcome, of course.

Something worth thinking about when looking at image recognition is things like colour , size , shape, texture etc. An extension to the this example might be to enable the net to recognise coloured characters but independent of the actual colour. If you always present 'A' in green and 'B' in blue and train it then when you come to test it might have just learned how to recognise the colours green and blue, then when you try and present a green 'B' it doesn't recognise it according to what you were thinking of. In this case you should present 'A' and 'B' in different colours.

In the classic tank hiding in jungle example a research team wanted to train a network to spot tanks hidden in a jungle. They went out an took pictures of tanks hiding in a jungle and pictures with no tanks. They trained the network and when they tested it the network worked very well. However to verify the network they went out a took more pictures and tested it again. This time it failed miserably. Why? For the training images the researchers took pictures of the tanks hiding in the jungle on sunny day and the ones where the tanks were not hiding on an overcast rainy day. The network had simply recognised that it was sunny or cloudy.

This example demonstrates that it's very important to apply good preprocessing techniques, in order to eliminate all the extraneous objects, colours and noise that could disturb the training of the network.

9 Applying Joone

9.1 Build your own first neural network

Even if the GUI Editor can be easily used to build, train and test neural networks, it's impractical to use within a real application to resolve your needs.

To really capture and use all the power of the core engine you need to write java code.
Indeed, as we'll see in the following paragraphs, the Editor can be used as a starting point to build a neural network - due to its user friendly interface - and then we can write java code to embed the resulting neural network into our own application.

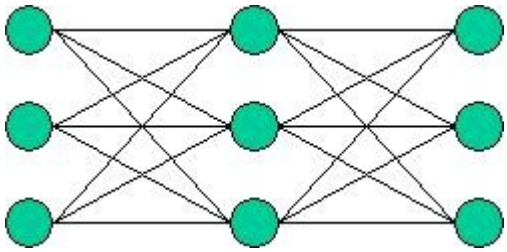
As of Joone 2.0, we have introduced an helper class, **`org.joone.helpers.factory.JooneTools`**, with which you can build & run a neural network in a very simple manner, by simply invoking some easy-to-use methods of the JooneTools class.
So you can use either the 'standard' method, or the JooneTools class, depending of the kind of network you need to build.

9.2 The standard API

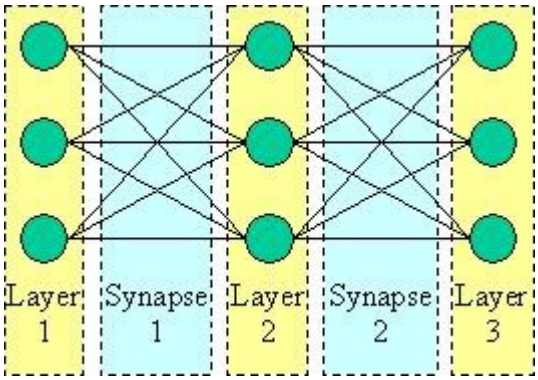
9.2.1 A simple (but useless) neural network

We will start by writing a simple toy neural network, and then will continue by building more complex architectures, until we'll be able to use almost all the features of the core engine.

Consider a feed-forward neural network composed of three layers like this:



To build this net with Joone, three Layer objects and two Synapse objects are required:




```
SigmoidLayer layer1 = new SigmoidLayer();
SigmoidLayer layer2 = new SigmoidLayer();
SigmoidLayer layer3 = new SygmoidLayer();
FullSynapse synapse1 = new FullSynapse();
FullSynapse synapse2 = new FullSynapse();
```

The SigmoidLayer objects and the FullSynapse objects are real implementations of the abstract Layer and Synapse objects.

Set the dimensions of the layers:

```
layer1.setRows(3);
layer2.setRows(3);
layer3.setRows(3);
```

Then complete the net, connecting the three layers with the synapses:

```
layer1.addOutputSynapse(synapse1);
layer2.addInputSynapse(synapse1);
layer2.addOutputSynapse(synapse2);
layer3.addInputSynapse(synapse2);
```

As you can see, each synapse is both the output synapse of one layer and the input synapse of the next layer in the net.
This simple net is ready, but it can't do any useful work because there are no components to read or write the data.
The next example shows how to build a real net that can be trained and used for a real problem.

9.2.2 A real implementation: the XOR problem.

Suppose a net to teach on the classical XOR problem is required.
In this example, the net has to learn the following XOR ‘truth table’:

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	0

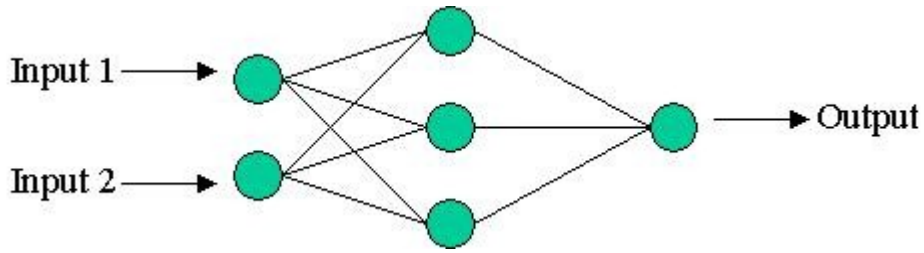
Firstly, a file containing these values is created:

```
0.0;0.0;0.0
0.0;1.0;1.0
1.0;0.0;1.0
1.0;1.0;0.0
```

Each column must be separated by a semicolon. The decimal point is not mandatory if the numbers are integer.
Write this file with a text editor and save it on the file system (for instance c:\joone\xor.txt in a Windows environment).
Now build a neural net that has the following three layers:

- An input layer with 2 neurons, to map the two inputs of the XOR function
- A hidden layer with 3 neurons, a good value to assure a fast convergence
- An output layer with 1 neuron, to represent the XOR function's output

as shown by the following figure:



First, create the three layers (using the sigmoid transfer function for the hidden and the output layers):

```
        LinearLayer input = new LinearLayer();
        SigmoidLayer hidden = new SigmoidLayer();
        SigmoidLayer output = new SigmoidLayer();
```

set their dimensions:

```
        input.setRows(2);
        hidden.setRows(3);
        output.setRows(1);
```

Now build the neural net connecting the layers by creating the two synapses using the FullSynapse class that connects all the neurons on its input with all the neurons on its output (see the above figure):

```
        FullSynapse synapse_IH = new FullSynapse(); /* Input -> Hidden conn. */
        FullSynapse synapse_HO = new FullSynapse(); /* Hidden -> Output conn. */
```

Next connect the input layer with the hidden layer:

```
        input.addOutputSynapse(synapse_IH);
        hidden.addInputSynapse(synapse_IH);
```

and then, the hidden layer with the output layer:

```
        hidden.addOutputSynapse(synapse_HO);
        output.addInputSynapse(synapse_HO);
```

Now we need a NeuralNet object that will contain all the Layers of the network:

```
NeuralNet nnet = new NeuralNet();
nnet.add(input, NeuralNet.INPUT_LAYER);
nnet.add(hidden, NeuralNet.HIDDEN_LAYER);
nnet.add(output, NeuralNet.OUTPUT_LAYER);
```

Now we'll set all the Monitor's parameters needed for the network to work:

```
Monitor monitor = nnet.getMonitor();
monitor.setLearningRate(0.8);
monitor.setMomentum(0.3);
```

The application registers itself as a monitor's listener, so it can receive the notifications of termination from the net. To do this, the application must implement the org.joone.engine.NeuralNetListener interface.

```
monitor.addNeuralNetListener(this);
```

Now define an input for the net, then create an org.joone.io.FileInputStream and give it all the parameters:

```
FileInputSynapse inputStream = new FileInputSynapse();
/* The first two columns contain the input values */
inputStream.setAdvancedColumnSelector("1,2");
/* This is the file that contains the input data */
inputStream.setInputFile(new File("c:\\joone\\XOR.txt"));
```

Next add the input synapse to the first layer. The input synapse extends the Synapse object, so it can be attached to a layer like a synapse.

```
  input.addInputSynapse(inputStream);
```

A neural net can learn from examples, so it needs to be provided it with the right responses.
For each input the net must be provided with the difference between the desired response and the actual response gave from the net. The org.joone.engine.learning.TeachingSynapse is the object that has this task:

```
TeachingSynapse trainer = new TeachingSynapse();
/* Setting of the file containing the desired responses, provided by a
FileInputSynapse */
FileInputSynapse samples = new FileInputSynapse();
samples.setInputFile(new File("c:\\joone\\XOR.txt"));
/* The output values are on the third column of the file */
samples.setAdvancedColumnSelector("3");
trainer.setDesired(samples);
```

The TeacherSynapse object extends the Synapse object.
So it can be added as the output of the last layer of the net.

```
output.addOutputSynapse(trainer);
/* We attach the teacher to the network */
nnet.setTeacher(trainer);
```

Set all the training parameters of the net:

```
monitor.setTrainingPatterns(4); /* # of rows in the input file */
monitor.setTotCicles(10000); /* How many times the net must be trained*/
```

```
    monitor.setLearning(true); /* The net must be trained */
    nnet.go(); /* The network starts the training phase */
```

Here is an example describing how to handle the `netStopped` and `cicleTerminated` events.

Remember:
To be notified, the main application must implement the `org.joone.NeuralNetListener` interface and must be registered to the Monitor object by calling the `Monitor.addNeuralNetListener(this)` method.

```
public void netStopped(NeuralNetEvent e) {
    System.out.println("Training finished");
}

public void cicleTerminated(NeuralNetEvent e) {
    Monitor mon = (Monitor)e.getSource();
    long c = mon.getCurrentCicle();
    /* We want print the results every 100 epochs */
    if (c % 100 == 0)
        System.out.println(c + " epochs remaining - RMSE = " +
              mon.getGlobalError());
}
```

(Many examples showing the above technique can be found in the org.joone.samples.engine.xor package of the source distribution)

9.3 Saving and restoring a neural network

To have the possibility of reusing a neural network built with Joone, we need to save it in a serialized format. To accomplish this goal, all the core elements of the engine implement the *Serializable* interface, permitting a neural network to be saved in a byte stream, to store it on the file system or data base, or transport it on remote machines using any wired or wireless protocol.

A simple way to save a neural network is to serialize the entire NeuralNet by using an ObjectOutputStream object, like illustrated in the following example that extends the XOR java class:

```
public void saveNeuralNet(String fileName) {
try {
      FileOutputStream stream = new FileOutputStream(fileName);
      ObjectOutputStream out = new ObjectOutputStream(stream);
      out.writeObject(nnet);
      out.close();
    } catch (Exception excp) {
         excp.printStackTrace();
    }
}
```

The writeObject method recursively saves all the objects contained in the non-transient variables of the serialized class, also avoiding having to store the same object's instance twice in case it is referenced by two separated objects – for instance a synapse connecting two layers.

We can later restore the above neural network using the following code:

```
public NeuralNet restoreNeuralNet(String filename) {
try {
      FileInputStream stream = new FileInputStream(fileName);
      ObjectInputStream inp = new ObjectInputStream(stream);
      return (NeuralNet)inp.readObject();
    } catch (Exception excp) {
         excp.printStackTrace();
         return null;
    }
}
```

As you can see, the above code is generic, as it doesn't depend on the internal structure of the saved/restored neural network.

Due to this motive, we have written a utility class, **org.joone.net.NeuralNetLoader**, that serves to reload a saved NeuralNet object, avoiding to write the above code whenever we need to load a serialized neural network.
It's very easy to use it:

```
      /* We need just to provide the serialized NN file name */
      NeuralNetLoader loader = new NeuralNetLoader("/somepath/myNet.snet");
      NeuralNet myNet = loader.getNeuralNet();
```

```
        . . .
```

so, by using only the above two simple lines of code, we're able to load in memory whatever serialized NeuralNet object, independently from its internal architecture.

9.4 Using the outcome of a neural network

After having learned how to train and save/restore a neural network, we will see how we can use the resulting patterns from a trained neural network.
To do this, we must use an object inherited from the OutputStreamSynapse class, so that we will be able to manage all the output patterns of a neural network for both the following two cases:

1. User's needs: to permit a user to read the results of a neural network, we must be able to write them onto a file, in some useful format, for instance, in ASCII format.
2. Application's needs: to permit an embedding application to read the results of a neural network, we must be able to write them onto a memory buffer – a 2D array of type double, for instance – and to read them automatically at the end of the elaboration.

Note: The examples shown in the following two chapters use the serialized form of the XOR neural network. To obtain that file, you must first create the XOR neural network with the editor, as illustrated in the GUI Editor User Guide, and export it using the File->Export menu item.

9.4.1 Writing the results to an output file

The first example we will see is about how to write the results of a neural network into an ASCII file, so a user can read and use it in practice.
To do this, we will use a FileOutputSynapse object, attaching it as the output of the last layer of the neural network. Assume that we have saved the XOR neural net from the previous example in a serialized form named 'xor.snet' so we can use it by simply loading it from the file system and attaching to its last layer the output synapse.
First of all, we write the code necessary to read a serialized NeuralNet object from an external application:

```
NeuralNet restoreNeuralNet(String fileName) {
      NeuralNetLoader loader = new NeuralNetLoader(fileName);
      NeuralNet nnet = loader.getNeuralNet();
      return nnet;
}
```

then we write the code to use the restored neural network:

```
NeuralNet xorNNet = this.restoreNeuralNet("/somepath/xor.snet");
if (xorNNet != null) {
      // we get the output layer
      Layer output = xorNNet.getOutputLayer();
      // we create an output synapse
      FileOutputSynapse fileOutput = new FileOutputSynapse();
      FileOutput.setFileName("/somepath/xor_out.txt");
      // we attach the output synapse to the last layer of the NN
      output.addOutputSynapse(fileOutput);
      // we run the neural network only once (1 cycle) in recall mode
      xorNNet.getMonitor().setTotCicles = 1;
```

```
      xorNNet.getMonitor().setLearning(false);
      xorNNet.go();
}
```

After the above execution, we can print out the obtained file, and, if the net is correctly trained, we will see a content like this:

```
0.016968769233825207
0.9798790621933134
0.9797402885436198
0.024205151360285334
```

This demonstrates the correctness of the previous training cycles.

9.4.2 Getting the results into an array

We now will see the use of a neural network from an embedding application that needs to use its results. The obvious approach in this case is to obtain the result of the recall phase into an array of doubles, so the external application can use it as needed.

We will see two usages of a trained neural network:

1. **The test of a net using a set of predefined patterns**; in this case we want to interrogate the net with several patterns, all collected before to query the net
2. **The test of a net using only one input pattern**; in this case we need to interrogate the net with a pattern provided by an external asynchronous source of data

We will see an example of both the above methods.

9.4.2.1 Using multiple input patterns

To accomplish this goal we will use the org.joone.io.MemoryOutputSynapse object, as illustrated in the following example.
Look at the following code:

```
    // The input array used for this example
    private double[][] inputArray = { {0, 0}, {0, 1}, {1, 0}, {1, 1} };

    private void Go(String fileName) {
        // We load the serialized XOR neural net
        NeuralNet xor = restoreNeuralNet(fileName);
        if (xor != null) {
            /* We get the first layer of the net (the input layer),
               then remove all the input synapses attached to it
               and attach a MemoryInputSynapse */
            Layer input = xor.getInputLayer();
            input.removeAllInputs();
            MemoryInputSynapse memInp = new MemoryInputSynapse();
            memInp.setFirstRow(1);
            memInp.setAdvancedColumnSelector("1,2");
            input.addInputSynapse(memInp);
```

```
            memInp.setInputArray(inputArray);

            /* We get the last layer of the net (the output layer),
               then remove all the output synapses attached to it
               and attach a MemoryOutputSynapse */
             Layer output = xor.getOutputLayer();
            // Remove all the output synapses attached to it...
            output.removeAllOutputs();
            //...and attach a MemoryOutputSynapse
            MemoryOutputSynapse memOut = new MemoryOutputSynapse();
            output.addOutputSynapse(memOut);
            // Now we interrogate the net
            xor.getMonitor().setTotCicles(1);
            xor.getMonitor().setTrainingPatterns(4);
            xor.getMonitor().setLearning(false);
            xor.go();
            for (int i=0; i < 4; ++i) {
                // Read the next pattern and print out it
                double[] pattern = memOut.getNextPattern();
                System.out.println("Output Pattern #"+(i+1)+" = "+pattern[0]);
            }
            xor.stop();
            System.out.println("Finished");        }
    }
```

As illustrated in the above code, we load the serialized neural net (using the same `restoreNeuralNet` method used in the previous chapter), and then we attach a MemoryInputSynapse to its input layer and a MemoryOutputSynapse to its output layer.
Before that, we have removed all the I/O components of the neural network, to be not aware of the I/O components used in the editor to train the net.
This is a valid example about how to dynamically modify a serialized neural network to be used in a different environment respect to that used for its design and training.

To provide the neural network with the input patterns, we must call the MemoryInputSynapse.setInputArray method, passing a predefined 2D array of double.
To get the resulting patterns from the recall phase we call the MemoryOutputSynapse.getNextPattern method; this method waits for the next output pattern from the net, returning an array of doubles containing the response of the neural network.
This call is made for each input pattern provided to the net.

The above code must be written in the embedding application, and to simulate this situation, we can call it from a `main()` method:

```
    public static void main(String[] args) {
        EmbeddedXOR xor = new EmbeddedXOR();
        xor.Go("org/joone/samples/engine/xor/xor.snet");
    }
```

The complete source code of this example is contained in the EmbeddedXOR.java file in the org.joone.samples.xor package.

9.4.2.2 Using only one input pattern

We now will see how to interrogate the net using only an input pattern.
We will show only the differences respect to the previous example:

```
private void Go(String fileName) {
    // We load the serialized XOR neural net
    NeuralNet xor = restoreNeuralNet(fileName);
    if (xor != null) {
        /* We get the first layer of the net (the input layer),
           then remove all the input synapses attached to it
           and attach a DirectSynapse */
        Layer input = xor.getInputLayer();
        input.removeAllInputs();
        DirectSynapse memInp = new DirectSynapse();
        input.addInputSynapse(memInp);
        ...
        /* We get the last layer of the net (the output layer),
           then remove all the output synapses attached to it
           and attach a DirectSynapse */
        Layer output = xor.getOutputLayer();
        output.removeAllOutputs();
        DirectSynapse memOut = new DirectSynapse();
        ...
```

As you can read, we now use both as input and output a DirectSynapse instead of the MemoryInputSynapse object.

What are the differences?

1. The DirectSynapse object is not a I/O component, as it doesn't inherit the StreamInputSynapse class
2. Consequently, it doesn't call the Monitor.nextStep method, so the neural network is not more controlled by the Monitor's parameters (see the Chapter 3 to better understand these concepts). Now the embedding application is responsible of the control of the neural network (it must know when to start and stop it), whereas during the training phase the start and stop actions were determined by the parameters of the Monitor object, being that process not supervised (remember that a neural network can be trained on remote machines without a central control).
3. For the same reasons, we don't need to set the 'TotCycles' and 'Patterns' parameters of the Monitor object.

Thus, to interrogate the net we can just write, after having invoked the *NeuralNet.start* method:

```
        xor.go(); // start the network
        for (int i=0; i < 4; ++i) {
            // Prepare the next input pattern
            Pattern iPattern = new Pattern(inputArray[i]);
            iPattern.setCount(1);
            // Interrogate the net
            memInp.fwdPut(iPattern);
            // Read the output pattern and print out it
            Pattern pattern = memOut.fwdGet();
            System.out.println("Output#"+(i+1)+" = "+pattern.getArray()[0]);
        }
```

In the above code we give the net only one pattern for each query, using the `DirectSynapse.fwdPut` method (note that this method accepts a Pattern object). As in the previous example, to retrieve the output pattern we call the `MemoryOutputSynapse.getNextPattern` method.

The complete source code of this example is contained in the ImmediateEmbeddedXOR.java file in the org.joone.samples.xor package.

9.5 Controlling the training of a neural network

9.5.1 Controlling the RMSE

In most cases it's very useful to control the behavior of a neural network at run time.
One of these cases could be represented by the necessity to stop the training of a neural network when its global error (RMSE) goes below a given value.

As probably you have noticed, in Joone there isn't any internal predefined mechanism to stop a neural network before the last training cycle is reached[6], hence it would be a wasting of time to continue to train the neural network after the RMSE became acceptable for our purposes.

The core engine comes in our aid by providing a notification mechanism based on events raised at the happening of certain facts.
The behavior of a neural network can be controlled by writing code in response of those neural network events; the code must be written into the corresponding event handler.
As already described in a previous chapter, there are four neural networks events that are raised in correspondence of the following actions:

- netStarted
- netStopped
- cycleTerminated
- errorChanged

The last two are denominated '*cyclic events*', and they are what we need to control the behavior of a neural network during its training (or querying) cycles.

If we need to stop the neural network when the RMSE reaches a given value, we can write the following code into the errorChanged event handler:

```
    public void errorChanged(NeuralNetEvent e) {
        Monitor mon = (Monitor)e.getSource();
        if (mon.getGlobalError() <= givenValue)
```

[6] Indeed in Joone there exists a component named ErrorBasedTerminator, that stops the network when a predefined RMSE value is reached. We don't use it here because the aim of this paragraph is to demonstrate how to handle the NeuralNet events.

```
            nnet.stop();
    }
```

We could also use this technique to write to the output console the current rmse every predetermined number of cycles, as described in the following sample code:

```
    public void cicleTerminated(NeuralNetEvent e) {
        Monitor mon = (Monitor)e.getSource();
        long c = mon.getTotCicles() - mon.getCurrentCicle();

        /* We want to print the result only every 1000 cycles */
        if ((c % 1000) == 0)
            System.out.println("Cycle:"+c+" RMSE = " + mon.getGlobalError());
    }
```

As you can see, by calling the `NeuralNetEvent.getSource()` method, we can obtain a pointer to the Monitor object of the current neural network, thanks to which we can control (almost) any aspect of the running neural network.

Important note: because all the above events are called synchronously by the threads running the neural network, avoid to make CPU intensive tasks within the event handler code. If you need to make some long elaboration, it would be better to instantiate a new thread, where the task could be executed without affecting the running of the neural network.

9.5.2 Cross Validation

Thanks to the possibility to execute java code in response of any events of the neural network, we can perform any kind of task, even if very complicated, as, for instance, the validation of a neural network 'on the fly' during the training phase, without the necessity to stop that phase.

To do it, we need to use two **LearningSwitch** components, along with some code executed in response of the **cicleTerminated** event.

In the example shown here we'll explain also some good programming techniques used to write a more readable and robust code, enhancing the code reuse.

First of all, as we need to repeat the same configuration (i.e. the chain input-->switch<--desired, as described in the chapter 4) both for the training and the desired input data, we'll write a generalized piece of code where we'll initialize all the components needed to perform our task:

```
    /** Creates a FileInputSynapse */
   private FileInputSynapse createInput(String name, int firstRow, int firstCol,
int lastCol) {
        FileInputSynapse input = new FileInputSynapse();
        input.setFileName(name);
        input.setFirstRow(firstRow);
        if (firstCol != lastCol)
            input.setAdvancedColumnSelector(firstCol+"-"+lastCol);
```

```
        else
            input.setAdvancedColumnSelector(Integer.toString(firstCol));

        // We normalize the input data in the range 0 - 1
        NormalizerPlugIn norm = new NormalizerPlugIn();
        if (firstCol != lastCol) {
            String ass = "1-"+Integer.toString(lastCol-firstCol+1);
            norm.setAdvancedSerieSelector(ass);
            }
        else
            norm.setAdvancedSerieSelector("1");
        input.setPlugIn(norm);
        return input;
    }
```

The above method creates and returns a FileInputSynapse with attached a NormalizerPlugin, simply by receiving as parameters the input file name, the first row, the first and last columns from which we must start to read the input data.

After that, we need to write a routine able to build the chain *inputSynapse --> LearningSwitch <-- DesiredSynapse*:

```
    /* Creates a LearningSwitch and attach to it both the training and
       the desired input synapses */
    private LearningSwitch createSwitch(StreamInputSynapse IT, StreamInputSynapse
IV) {
        LearningSwitch lsw = new LearningSwitch();
        lsw.addTrainingSet(IT);
        lsw.addValidationSet(IV);
        return lsw;
    }
```

At this point we can simply call the above two methods to build the input and desired data components:

```
        /* Creates all the required input data sets:
         * ITdata = input training data set
         * IVdata = input validation data set
         * DTdata = desired training data set
         * DVdata = desired validation data set
         */
        FileInputSynapse ITdata = this.createInput(path+"/data.txt",1,2,14);
        FileInputSynapse IVdata = this.createInput(path+"/data.txt",131,2,14);
        FileInputSynapse DTdata = this.createInput(path+"/data.txt",1,1,1);
        FileInputSynapse DVdata = this.createInput(path+"/data.txt",131,1,1);

        /* Creates and attach the input learning switch */
        LearningSwitch Ilsw = this.createSwitch(ITdata, IVdata);
        InputLayer.addInputSynapse(Ilsw);

        /* Creates and attach the desired learning switch */
        LearningSwitch Dlsw = this.createSwitch(DTdata, DVdata);
        TeachingSynapse ts = new TeachingSynapse(); // The teacher of the net
        ts.setDesired(Dlsw);
        OutputLayer.addOutputSynapse(ts);
```

In the above example we have used the first 130 rows as training patterns, and the remaining rows as validation data. Moreover, we use the columns from 2 to 14 as input data, and the first one as target value.

As you can see in the above code, at the end we have attached the input and desired switches to the input layer and the teacher respectively (we have omitted the code to build the layers of the neural network, but you should be able to do it yourself without problems).

Now we must add the code needed to perform the validation of the neural network at end of every training epoch.

Of course, that code must be written into the cicleTerminated event handler:

```
    public void cicleTerminated(NeuralNetEvent e) {
        Monitor mon = (Monitor)e.getSource();

        // Prints out the current epoch and the training error
        int cycle = mon.getCurrentCicle()+1;
        if (cycle % 200 == 0) { // We validate the net every 200 cycles
          System.out.println("Epoch #"+(mon.getTotCicles() - cycle));
          System.out.println("   Training Error:"+mon.getGlobalError());

          // Creates a copy of the neural network
          net.getMonitor().setExporting(true);
          NeuralNet newNet = net.cloneNet();
          net.getMonitor().setExporting(false);

          // Cleans the old listeners
          // This is a fundamental action to avoid that the validating net
          // calls the cicleTerminated method of this class
          newNet.removeAllListeners();

          // Set all the parameters for the validation
          NeuralNetValidator nnv = new NeuralNetValidator(newNet);
          nnv.addValidationListener(this);
          nnv.start();  // Validates the net
        }
    }
```

Even if the code is rather self-explaining, we want to emphasize the following aspects:

You can notice that the main neural network is not stopped during the validation phase, and this is possible thanks to the cloning capacity of the NeuralNet object; as you can see, in fact, we validate a *cloned copy* of the neural network, while the main neural network continues to be trained.
This offers some advantages, because we perform in parallel the validation phase, being so able to take advantage of the presence of a multiprocessor architecture.

To perform the validation task we use the **NeuralNetValidator** object. It runs on a separate thread and notifies the main application by issuing a netValidated event (to be notified, the main application must implement the NeuralValidationListener interface).

The following code illustrates what we do in response of a netValidated event:

```
/* Validation Event */
public void netValidated(NeuralValidationEvent event) {
  // Shows the RMSE at the end of the cycle
```

```
   NeuralNet NN = (NeuralNet)event.getSource();
   System.out.println("   Validation Error: "+NN.getMonitor().getGlobalError());
 }
```

As you can see, the variable passed as parameter of the method contains a pointer to the validated neural network (that one that we have cloned in the previous code), so we're able to access to all the parameters of the validation task from within the caller main application (in this example we use it to get the validation RMSE).

If you want to try yourself the above example, you can find the complete code into the *org.joone.samples.engine.validation.SimpleValidationSample* class, and if you run it, you'll get a result like the following:

```
Epoch #200
    Training Error:   0.03634410057758484
    Validation Error: 0.08310312916100844
Epoch #400
    Training Error:   0.023295226557687492
    Validation Error: 0.07643178777353665
Epoch #600
    Training Error:   0.017832470096609952
    Validation Error: 0.07457234059641271
...
```

In this example we have just used the validated neural network to get and print the validation error, but you could perform whatever task as, for instance, to save in serialized format each validated neural network, or only those having a RMSE lower than a predefined value, in order to be able to perform a selection of the best neural networks (i.e. those having the best generalization capacity) at the end of the training phase.

A good technique could be represented by the implementation of the following algorithm, also known as “Early Stopping”:

1. When we start the main network, a variable named lastRMSE must be set to a high value, say 999
2. In response to the netValidated event, if the returned validation RMSE < lastRMSE, then save the returned network and let lastRMSE = RMSE
3. Otherwise, do not save the network and stop the training phase. The last saved network is the best one.

When in the step 2 we notice that the validation error begins to increase, then we're sure that the last saved network is the best one (e.g. the neural network with the best generalization error), hence we stop the training phase.

Note: This technique is very powerful when used in conjunction with the distributed training environment, where you can run several copies of the same neural network (each one initialized with different random weights) by using different machines connected to a LAN, augmenting in this manner the probability to find a neural network having very good performances in terms of generalization capacity.

9.6 The JooneTools helper class

JooneTools is a class that exposes many useful static methods to build and run a neural network by hiding the complexity of the core engine's API.
It can be used in a lot of circumstances, whenever the network you need to build belongs to one of the standard architectures supported by JooneTools (feed forward and SOM networks at the moment), and when the training or interrogation phases must be performed without any particular customization. By reading the following paragraphs, and reading the JooneTools API javadoc, you'll be able to understand when and how to use it.

9.6.1 Building & runnning a simple feed forward neural network

By using JooneTools, you can easily build, train and interrogate a feed forward neural network simply writing 3 (yes, *three* :-) rows of code!
Look at the following example:

```
// Create an MLP network with 3 layers [2,2,1 nodes] with a logistic output layer
NeuralNet nnet = JooneTools.create_standard(new int[]{2,2,1},
                  JooneTools.LOGISTIC);

// Train the network for 5000 epochs, or until the rmse < 0.01
double rmse = JooneTools.train(nnet, inputArray, desiredArray,
                  5000,       // Max epochs
                  0.01,       // Min RMSE
                  0,          // Epochs between ouput reports
                  null,       // Std Output
                  false       // Asynchronous mode
              );

// Interrogate the network
double[] output = JooneTools.interrogate(nnet, testArray);
```

Let's explain the methods used:

JooneTools.create_standard: this method creates and returns a new feed-forward neural network. The number of layers will be equal to the size of the array of integers passed as the first parameter; each element of the array indicates the nodes (or rows) contained in each layer. The first Layer will be always composed by a LinearLayer, the hidden nodes will be composed by SigmoidLayers, while the output layer kind is determined by the second parameter of the method, that can be one of the constants indicated in the following table:

Constant used	**Kind of output layer**	**Problem to resolve**
JooneTools.LINEAR	LinearLayer	Function approximation
JooneTools.LOGISTIC	SigmoidLayer	Binary classification
JooneTools.SOFTMAX	SoftmaxLayer	1 of C classification

You'll choose the kind of output layer depending on the kind of problem you need to resolve, as indicated in the table.

JooneTools.train: this method trains a network in supervised mode, according to some parameters (listed in the order expected by the method):

1. The neural network to train; it must contain only the input, hidden and output layers, without any I/O components attached (like that one returned by the create_standard method, for example).
2. A 2D array of doubles containing the training input data. The array must have a number of columns equal to the number of network's input nodes and a number of rows equal to the number of training patterns to use.
3. A 2D array of doubles containing the training desired data. The array must have a number of columns equal to the number of network's output nodes and a number of rows equal to the number of training patterns to use.
4. The number of (max) training epochs.
5. The min RMSE; the network will be trained until its rmse will be greater than this parameter (if >0), otherwise the training will continue for the number of epochs indicated in the previous parameter.
6. The number of epochs between two notifications (see the next parameter). 0 if no notifications desired.
7. A pointer to the object that will receive the network's notifications. It can implement either a NeuralNetListener, or a PrintStream class, depending on the kind of the notification we want to receive. If the object is a NeuralNetListener, the corresponding methods will be invoked, otherwise, in case of a PrintStream class (like System.out, for instance), a preformatted text will be written. In both the cases, the interval of epochs between two notifications is determined by the content of the previous parameter. Null if no notifications needed.
8. A boolean indicating if the training must be executed in asynchronous mode. If true, the method will return immediately and the network will be trained in background, within a separate thread. If false, the method will return only after the training is terminated.

While almost all the above parameters have a clear meaning, maybe the 6th and 7th need a deeper explanation.
JooneTools permits to monitor the training progress in two manners:

By using a NeuralNetListener: this is the classic method, where the caller application needs to declare and pass a NeuralNetListener class, as in the following example:

```
NeuralNetListener listener = new NeuralNetListener() {
        public void netStarted(NeuralNetEvent e) { ... }
        public void cicleTerminated(NeuralNetEvent e) { ... }
        public void errorChanged(NeuralNetEvent e) { ... }
        public void netStopped(NeuralNetEvent e) { ... }
        public void netStoppedError(NeuralNetEvent e,String error) { ... }
}
double rmse = JooneTools.train(nnet, inputArray, desiredArray,
                    5000,       // Max epochs
                    0.01,       // Min RMSE
```

```
                        100,        // Epochs between notifications
                        listener,   // Notifications listener
                        false       // Asynchronous mode
                );
```

In the above example the declared listener will be used, and its cyclic methods, like cycleTerminated and errorChanged, will be invoked each 100 epochs.

By using a PrintStream class: when we don't need to execute custom code in response of a network's event, but we want anyway to be informed about the training progress, we can pass as listener a simple PrintStream class (like, for instance, System.out, if we want the messages printed on the console).
Look at the following example:

```
    double rmse = JooneTools.train(nnet, inputArray, desiredArray,
                        5000,       // Max epochs
                        0.01,       // Min RMSE
                        200,        // Epochs between notifications
                        System.out, // Output to the system console
                        false       // Asynchronous mode
                );
```

In this case all the network's events will be notified on the system console with a periodicity of 200 epochs:

```
Network started
Epoch n.200 terminated - rmse: 0.3552344523359257
Epoch n.400 terminated - rmse: 0.09979108423932816
Epoch n.600 terminated - rmse: 0.038605717897835144
...
Network stopped
```

Of course you can use whatever else class that extends PrintStream, in order to direct the output messages to a different media.

The last method used is
JooneTools.interrogate: as the name indicates, this method is used to interrogate a trained network using a single input pattern. The method returns an array of double containing the outcome of the neural network. As parameters, it accept the NeuralNet object to interrogate, and an array of double containing the input data to use. The input array must have as many elements as the size of the output layer of the network.

In the `org.joone.samples.engine.helpers.XOR_using_helpers` class you can find a complete example illustrating the use of JooneTools to build, train and interrogate a XOR network.

9.6.2 The JooneTools I/O helper methods

As previously illustrated, many methods of JooneTools expect an array of double as input data. In order to easily extract such an array from an input stream, in JooneTools we'll find the method getDataOnStream, that can be used in the following manner:

```
    // Prepare the training and testing data set
    FileInputSynapse fileIn = new FileInputSynapse();
    fileIn.setInputFile(new File(fileName));
    fileIn.setAdvancedColumnSelector("1-14");

    // Input data normalized between -1 and +1
    NormalizerPlugIn normIn = new NormalizerPlugIn();
    normIn.setAdvancedSerieSelector("2-14");
    normIn.setMin(-1);
    normIn.setMax(1);
    fileIn.addPlugIn(normIn);

    // Target data normalized between 0 and 1
    NormalizerPlugIn normDes = new NormalizerPlugIn();
    normDes.setAdvancedSerieSelector("1");
    fileIn.addPlugIn(normDes);

    // Extract the training data
    double[][] inputTrain = JooneTools.getDataFromStream(fileIn,
                              1, trainingRows, 2, 14);
    double[][] desiredTrain = JooneTools.getDataFromStream(fileIn,
                              1, trainingRows, 1, 1);

    // Extract the testing data
    double[][] inputTest = JooneTools.getDataFromStream(fileIn,
                              trainingRows+1, 178, 2, 14);
    double[][] desiredTest = JooneTools.getDataFromStream(fileIn,
                              trainingRows+1, 178, 1, 1);
```

In the above example we used a FileInputSynapse to read the input data, composed by 178 patterns, 14 columns each.
We use the columns from 2 to 14 as input, while the first column contains the desired data the network must learn to recognize.

After having normalized both the input and desired data by using two `NormalizerPlugins` (as already described in the previous chapters), we use the resulting `FileInputSynapse` as input parameter for the invocation of the `JooneTools.getDataFromStream` method, passing each time all the parameters needed to extract the input&desired arrays of data, both for training and testing phases.
Now, having extracted the corresponding four arrays of double, we can use them to train and interrogate the network, by comparing the results on the test data, as illustrated in the following code:

```
    // Train the network
    JooneTools.train(nnet, inputTrain, desiredTrain,
                5000,   // Max # of epochs
                0.010,  // Stop RMSE
                100,    // Epochs between output reports
                this,   // The listener
```

```
                false); // Runs in synch mode

...
    // And now compare the results on the test set
    double[][] out = JooneTools.compare(nnet, inputTest,
                                        desiredTest);
    System.out.println("Comparison of the last "+out.length+"
                        rows:");
    int cols = out[0].length/2;
    for (int i=0; i < out.length; ++i) {
        System.out.print("\nOutput: ");
        for (int x=0; x < cols; ++x) {
            System.out.print(out[i][x]+" ");
        }
        System.out.print("\tTarget: ");
        for (int x=cols; x < cols*2; ++x) {
            System.out.print(out[i][x]+" ");
        }
    }
```

By running the above example, (the complete source code is in org/joone/samples/engine/helpers/Validation_using_stream.java), you'll obtain an output like the following:

```
Comparion of the last 28 rows:

Output: 0.001644333042189837  Target: 0.0
Output: 9.292946600039575E-4  Target: 0.0
Output: 0.4855262175163008    Target: 0.5
Output: 0.9976350028550492    Target: 1.0
...
...
Output: 0.11104094434076456   Target: 0.5
Output: 0.6211928869659087    Target: 0.5
```

We have introduced here a new JooneTools method named '`compare`', using which you can easily extract both the response of the network and the target values within the same array, in order to be able to make comparisons between them.
The **`JooneTools.compare`** method, in fact, returns a 2D array of double containing the output+target data for each pattern (the resulting output array's number of columns is the double of the target array size).

9.6.3 Testing the performances of a network

Finally, you can also test the performances of a network (i.e. calculate the resulting RMSE for a specific input pattern) by using the **`JooneTools.test`** method. It accepts as parameters the NeuralNet object containing the network to test, the input test data and the corresponding desired data. All the data, as always, must be contained in an array of double, that you can obtain by

invoking the JooneTools.getDataFromStream method, as seen in the previous paragraph.
The test method returns a double indicating the RMSE obtained on the input patterns, compared with the given target data.
This method is very useful, for example, to calculate the generalization capacity of a network on unseen data.

9.6.4 Building unsupervised (SOM) networks

JooneTools permits to create unsupervised Self Organized Map network by exposing the `JooneTools.createUnsupervised` method.

It accepts two parameters:

- **nodes**: An array of integer containing 3 elements, having the following meaning:
  - nodes[0] = Rows of the input layer
  - nodes[1] = Width of the output map
  - nodes[2] = Height of the output map

- **outputType**: an integer indicating the kind of output layer we need. It can contain one of the following two constants:
  - `JooneTools.WTA` - SOM with a WinnerTakeAll output layer
  - `JooneTools.GAUSSIAN` - SOM with a Gaussian output layer

Once we have created the SOM network, we can train it by using the **`JooneTools.train_unsupervised`** method. See the JooneTools API javadoc to read about the parameters accepted by this method.

9.6.5 Loading and saving a network with JooneTools

JooneTools exposes, of course, also some methods to save/load easily a neural network. You can use the following methods:

Method name	**Purpose**
save(NeuralNet network, String fileName)	Saves a network to a file
save_toStream(NeuralNet nnet, OutputStream stream)	Saves a network to an OutputStream
load(String fileName)	Loads a network from a file
load_fromStream(InputStream stream)	Loads a network from an InputStream

The above methods save/load a network ONLY in java serialized format. The XML format is not still supported.

10 The LGPL Licence

GNU LESSER GENERAL PUBLIC LICENSE
Version 2.1, February 1999

[This is the first released version of the Lesser GPL. It also counts as the successor of the GNU Library Public License, version 2, hence the version number 2.1.]

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licenses are intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users.

This license, the Lesser General Public License, applies to some specially designated software packages--typically libraries--of the Free Software Foundation and other authors who decide to use it. You can use it too, but we suggest you first think carefully about whether this license or the ordinary General Public License is the better strategy to use in any particular case, based on the explanations below.

When we speak of free software, we are referring to freedom of use, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish); that you receive source code or can get it if you want it; that you can change the software and use pieces of it in new free programs; and that you are informed that you can do these things.

To protect your rights, we need to make restrictions that forbid distributors to deny you these rights or to ask you to surrender these rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link other code with the library, you must provide complete object files to the recipients, so that they can relink them with the library after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

We protect your rights with a two-step method: (1) we copyright the library, and (2) we offer you this license, which gives you legal permission to copy, distribute and/or modify the library.

To protect each distributor, we want to make it very clear that there is no warranty for the free library. Also, if the library is modified by someone else and passed on, the recipients should know that what they have is not the original version, so that the original author's reputation will not be affected by problems that might be introduced by others.

Finally, software patents pose a constant threat to the existence of any free program. We wish to make sure that a company cannot effectively restrict the users of a free program by obtaining a restrictive license from a patent holder. Therefore, we insist that any patent license obtained for a version of the library must be consistent with the full freedom of use specified in this license.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License. This license, the GNU Lesser General Public License, applies to certain designated libraries, and is quite different from the ordinary General Public License.

We use this license for certain libraries in order to permit linking those libraries into non-free programs.

When a program is linked with a library, whether statically or using a shared library, the combination of the two is legally speaking a combined work, a derivative of the original library. The ordinary General Public License therefore permits such linking only if the entire combination fits its criteria of freedom. The Lesser General Public License permits more lax criteria for linking other code with the library.

We call this license the "Lesser" General Public License because it does Less to protect the user's freedom than the ordinary General Public License. It also provides other free software developers Less of an advantage over competing non-free programs. These disadvantages are the reason we use the ordinary General Public License for many libraries. However, the Lesser license provides advantages in certain special circumstances.

For example, on rare occasions, there may be a special need to encourage the widest possible use of a certain library, so that it becomes a de-facto standard. To achieve this, non-free programs must be allowed to use the library. A more frequent case is that a free library does the same job as widely used non-free libraries. In this case, there is little to gain by limiting the free library to free software only, so we use the Lesser General Public License.

In other cases, permission to use a particular library in non-free programs enables a greater number of people to use a large body of free software. For example, permission to use the GNU C Library in non-free programs enables many more people to use the whole GNU operating system, as well as its variant, the GNU/Linux operating system.

Although the Lesser General Public License is Less protective of the users' freedom, it does ensure that the user of a program that is linked with the Library has the freedom and the wherewithal to run that program using a modified version of the Library.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the library" and a "work that uses the library". The former contains code derived from the library, whereas the latter must be combined with the library in order to run.

GNU LESSER GENERAL PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any software library or other program which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Lesser General Public License (also called "this License"). Each licensee is addressed as "you".

A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

"Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish

on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) The modified work must itself be a software library.

b) You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.

c) You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.

d) If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

(For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also combine or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)

b) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (1) uses at run time a copy of the library already present on the user's computer system, rather than copying library functions into the executable, and (2) will operate properly with a modified version of the library, if the user installs one, as long as the modified version is interface-compatible with the version that the work was made with.

c) Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.

d) If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above

specified materials from the same place.

e) Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the materials to be distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.

b) Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties with this License.

11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the

sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

13. The Free Software Foundation may publish revised and/or new versions of the Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Libraries

If you develop a new library, and you want it to be of the greatest possible use to the public, we recommend making it free software that everyone can redistribute and change. You can do so by permitting redistribution under these terms (or, alternatively, under the terms of the ordinary General Public License).

To apply these terms, attach the following notices to the library. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the library's name and a brief idea of what it does.>
Copyright (C) <year> <name of author>

```
    This library is free software; you can redistribute it and/or
    modify it under the terms of the GNU Lesser General Public
    License as published by the Free Software Foundation; either
    version 2.1 of the License, or (at your option) any later version.

    This library is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
    Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public
    License along with this library; if not, write to the Free Software
    Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

Also add information on how to contact you by electronic and paper mail.

You should also get your employer (if you work as a programmer) or your school, if any,
to sign a "copyright disclaimer" for the library, if necessary.  Here is a sample; alter
the names:

  Yoyodyne, Inc., hereby disclaims all copyright interest in the   library `Frob' (a
library for tweaking knobs) written by James Random Hacker.

  <signature of Ty Coon>, 1 April 1990
  Ty Coon, President of Vice

That's all there is to it!
```