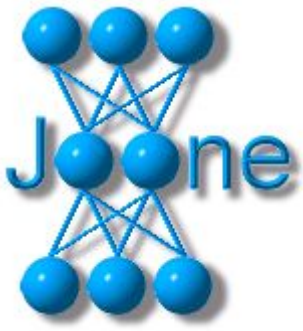


Java Object Oriented Neural Engine

The Complete Guide

All you need to know about Joone

http://www.joone.org
Paolo Marrone (*pmarrone@users.sourceforge.net*)

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 initialised neural network (NN) 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 can continue retraining the selected NNs until some final parameter is reached (i.e. a low RMSE value) like a distillation process. The best architecture is 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 was to make easily usable and distributable a trained NN 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' resulting network to its sales force, that they can use it on their mobile devices.
This is why Joone is serializable and remotely transportable using any wired and 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 Joone interesting and useful, I thank you for your interest to it.

Paolo Marrone

[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 *Portable Systems*:

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 product on the market, so Joone can gain a large market share and become the most used neural network framework.

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 for One-to-One marketing, etc.)

1.2.3 Portable systems

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, marketing 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 complete 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 explains 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 the 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 three persons that have contributed continuously for a long period of time, writing good java code, and also supporting me with very interesting proposals and suggestions (listed in alphabetical order):

Harry Glasgow
Julien Norman
Paul Sinclair

Thanks also to the following people that have collaborated to Joone in the past:

Mark Allen, Ka-Hing Cheung, Jan Erik Garshol, Jack Hawkins, Olivier Hussenet, Shen Linlin, Christian Ribeaud, Anat Rozenzon, Thomas Lionel Smets

---

Do you want to see your name listed above? Join us: any contribution is **always** welcome, therefore if you are interested to participate to Joone, contact me specifying what are your past experiences on artificial neural networks and java programming, and what you'd like to do for Joone - Write **only** if really interested to participate -

---

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
- **VisAD** http://www.ssec.wisc.edu/~billh/visad.html

A particular acknowledgment to:

- SourceForge.net, thanks to which all this has been possible
- **Nathan Hindley** who has designed and realized the amazing web site at http://www.joone.org (you can contact him at kelticdanor@iprimus.com.au)
- Zero G Software, Inc. for InstallAnywhere, the multiplatform 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 or later is available.
Due to his direct experience, or because he has received information from other users, the author can assure the compatibility of Joone with the following operating systems[2]:

- Linux
- Mac OSX
- Windows 2000
- Windows XP

About the memory requirement, it depends on the complexity of the neural network used, but generally the availability of at least 128MB 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 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
xalan.jar crimson.jar	The XML libraries	**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 following two are needed only if you use some specific feature of the core engine.

The last four 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

To permit to everyone to correctly install and run the GUI Editor, this is distributed in an auto-installing form. Using the ZeroG Software InstallAnywhere product we have prepared auto-installers for the following platforms[3]:

- Linux
- Windows
- Mac OSX

You don't need to be aware about the installation of the Java runtime environment, as all the installers are available both with and without an embedded java virtual machine (except for the Mac version, because on the OSX platform a suitable JVM is already installed).
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.

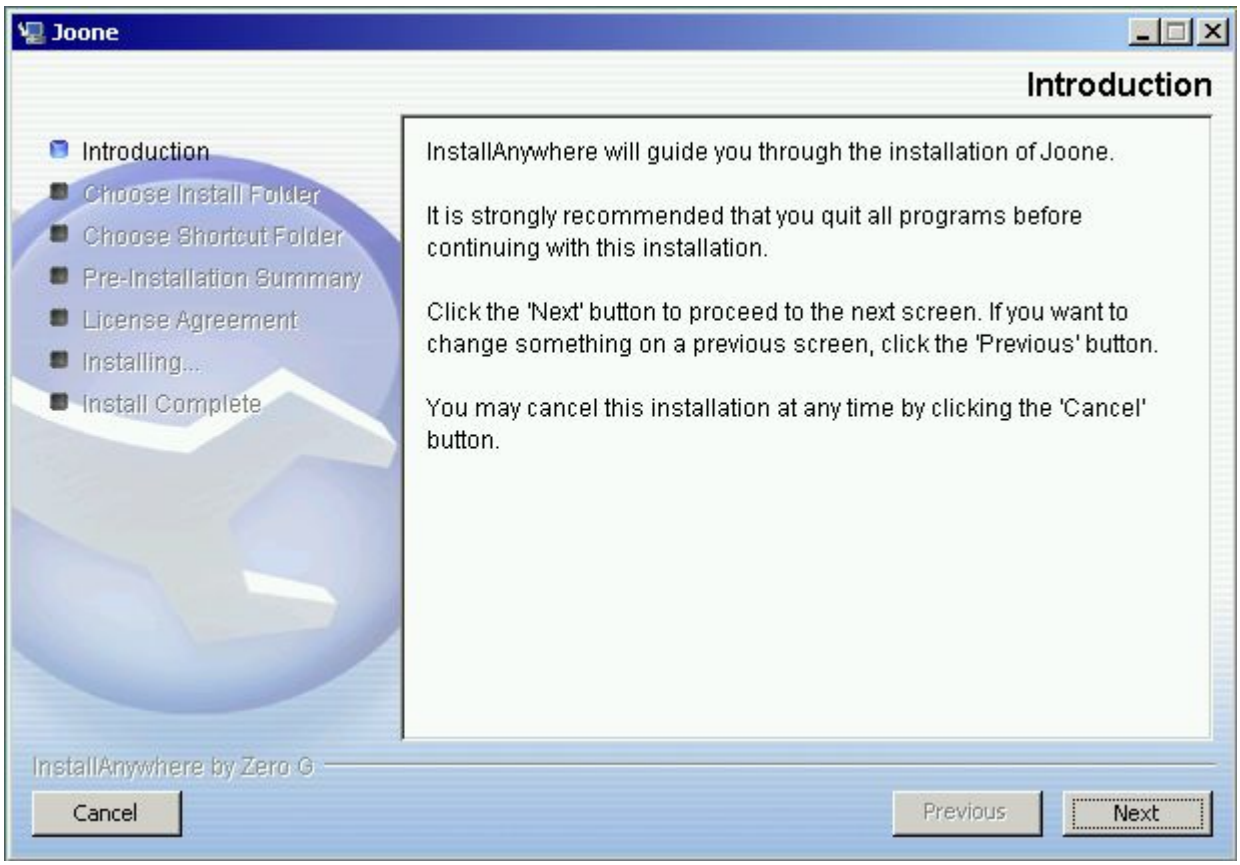
Windows Instructions:
After downloading, double-click JooneEditorX.Y.Z.exe
If you do not have a Java virtual machine installed, be sure to download the package which includes one.

Mac OS X Instructions:

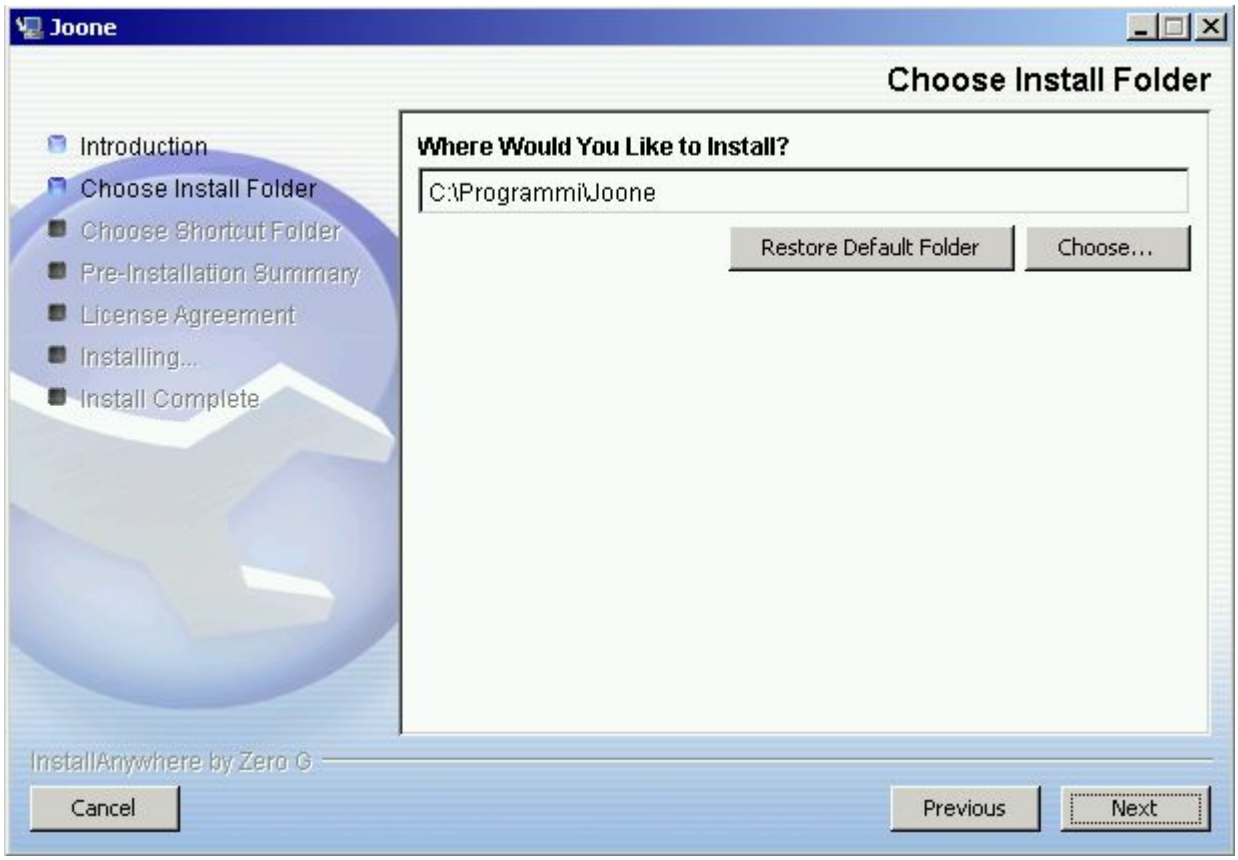
[3] InstallAnywhere is a registered trademark of Zero G Software, Inc.
Mac OS is a registered trademark of Apple Computer, Inc.
Solaris and Java are trademarks of Sun Microsystems, Inc.
Windows is a registered trademark of Microsoft Corporation.
All other marks are properties of their respective owners.

After downloading, double-click JooneEditorX.Y.Z.zip (Requires Mac OS X 10.0 or later).
The compressed installer should be recognized by Stuffit Expander and should automatically be expanded after downloading. If it is not expanded, you can expand it manually using StuffIt Expander 6.0 or later.
If you have any problems launching the installer once it has been expanded, make sure that the compressed installer was expanded using Stuffit Expander.

After the launch of the installer, you should see the following panel:



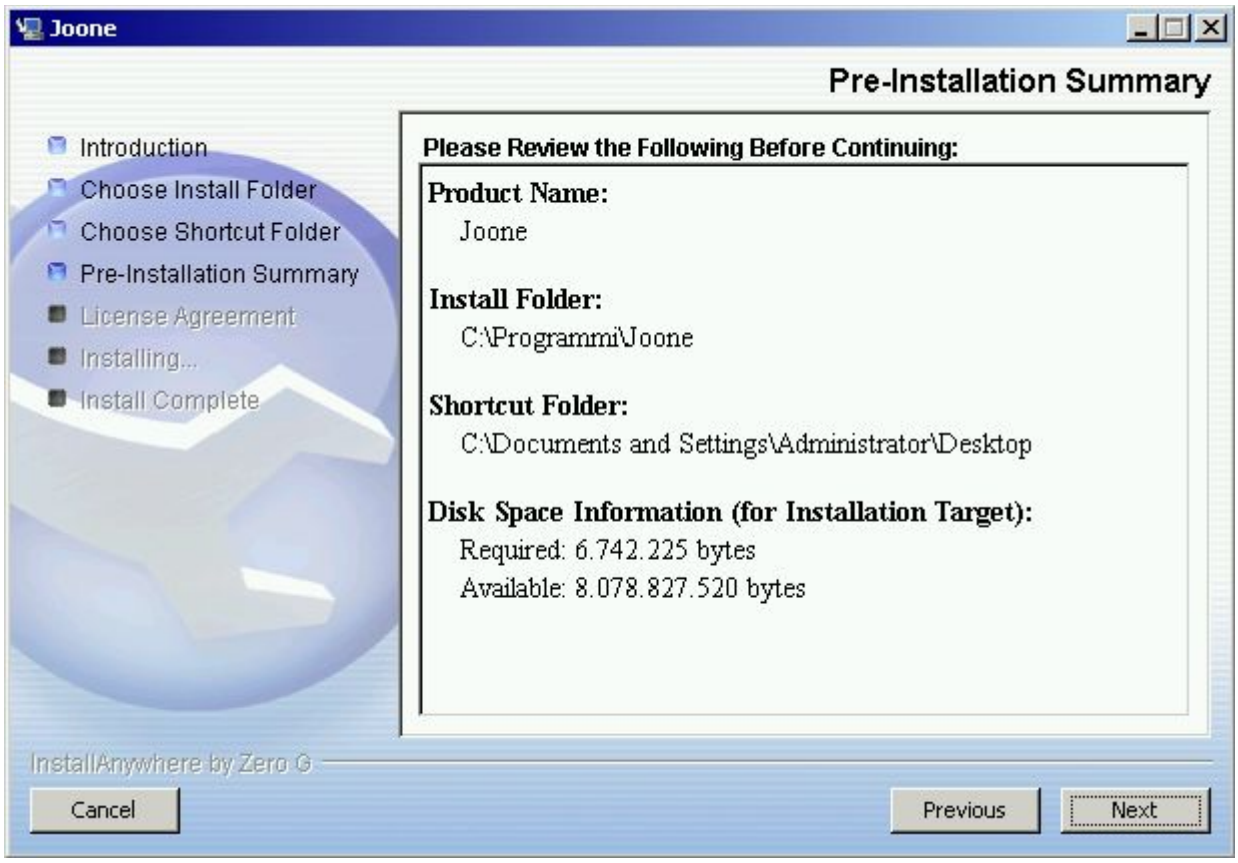
By clicking on the Next button you can advance in the installation process. In any moment, pressing the Cancel button, you can abort and exit from the installation.



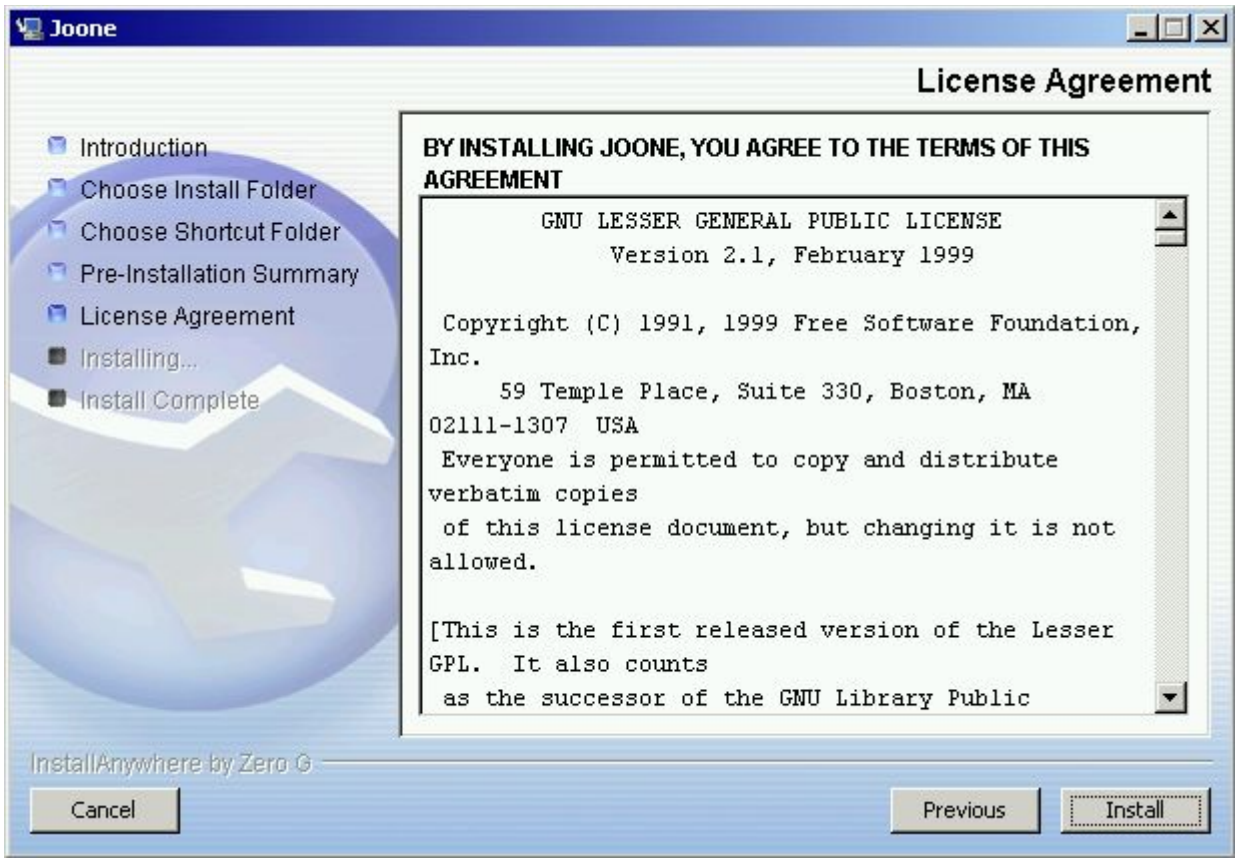
In this panel you must specify the directory where you want to install Joone.
The Choose button will open an explorer window, where you can make the choice, whereas using the ‘Restore Default Folder’ button you can reset the directory to its initial value.

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 one of the last Chapters 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'llshow 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.xip (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 /home/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@cvs.sourceforge.net:/cvsroot/joone login
cvs -z3 -d:pserver:anonymous@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 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 /home/joone) and save the file.

Assuming you have the Java JDK and ANT correctly installed and running (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:

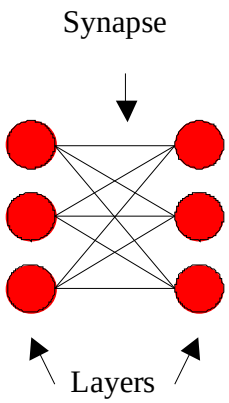
1. Put the <base_dir>/build directory and all the <base_dir>/lib/*.jar files on your classpath
2. Open a console and launch the following command: java org.joone.edit.JoonEdit

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. Recurrent connections from a layer to a previous one are not permitted. Consider the following figure:

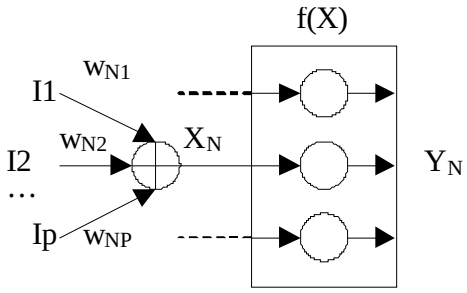


This is a sample FFNN with two layers connected with one synapse. Each layer is composed of a certain number of neurons, each of which have the same characteristics (transfer function, learning rate, etc).
A neural net built with Joone can be composed of whatever number of layers of different kinds of layer.
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

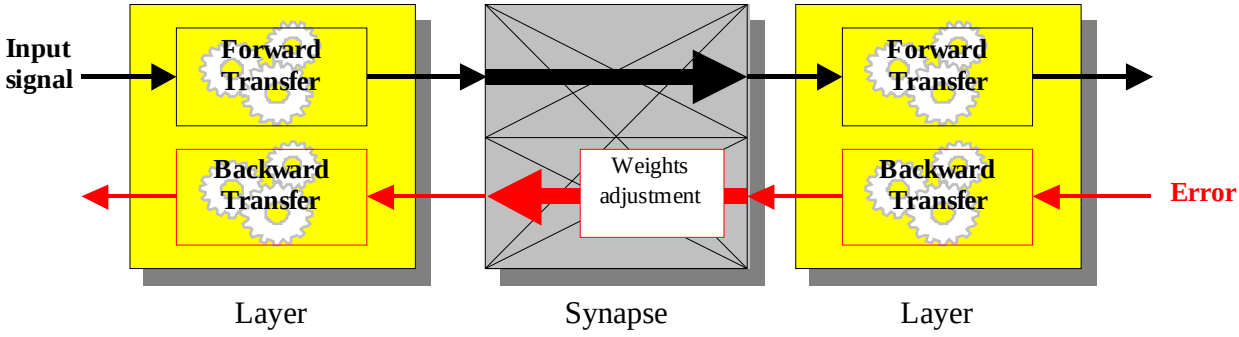
To ensure that it is possible to build whatever neural network architecture is required with Joone, a method to transfer the patterns through the net is required without the need of a central point of control.

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 scheme:



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

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:

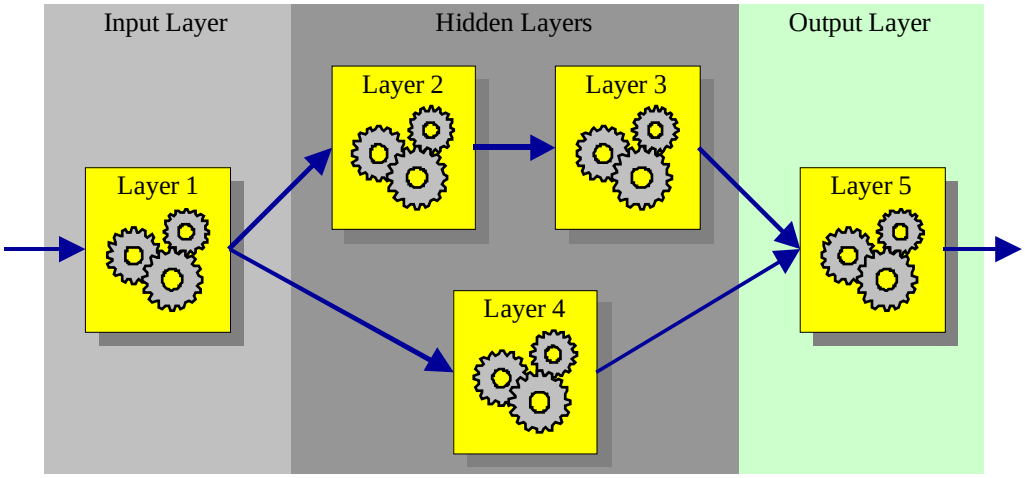


Each Joone component (both layers and synapses) has its own pre-built mechanisms to adjust the weights and biases according to the chosen learning algorithm.

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 in its output synapses, to which there are other layers connected running on separate threads, and so on.

Joone allows any kind of net to be built through 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.

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.

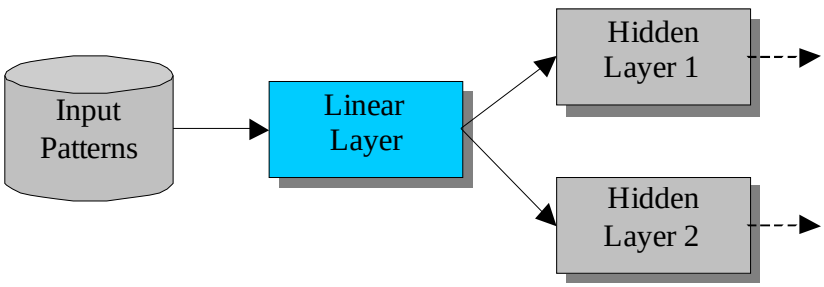
3.3.1 The Layers

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 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.

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 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

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

3.3.1.3 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

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

3.3.1.4 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.

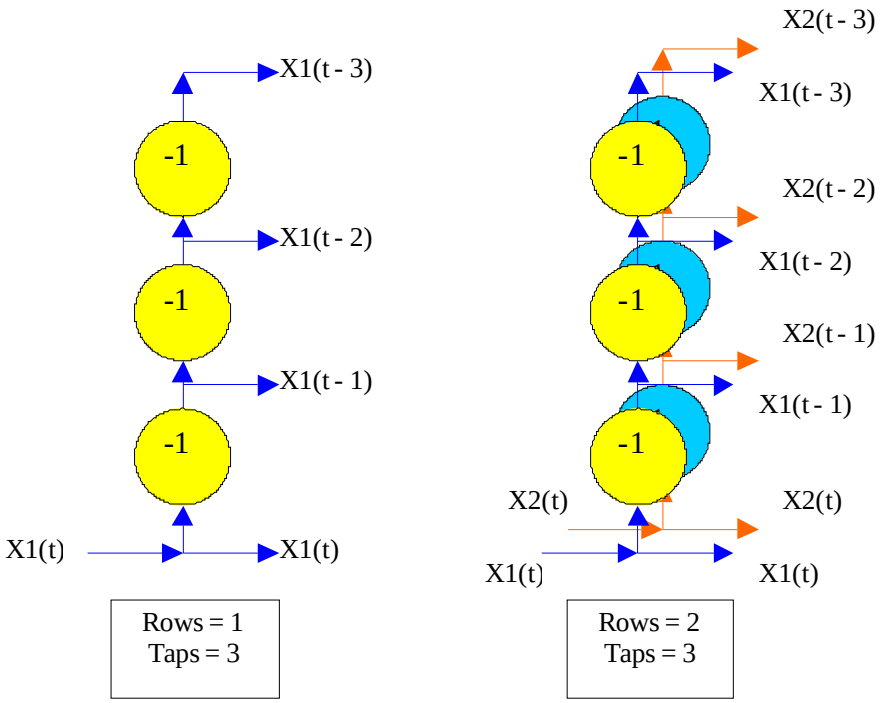
Transfer Function

$y = \log(1 + x)$ if $x \geq 0$
$y = \log(1 - x)$ if $x < 0$

3.3.1.5 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 the 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 sum signal Xn (t) to the output. During a 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

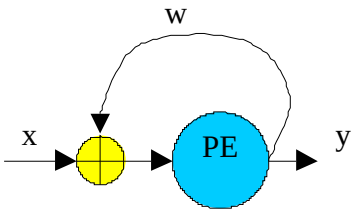
$y_N = x_{(t-N)}$

where: $0 < N \leq taps$

3.3.1.6 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

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

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

3.3.1.7 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 detecting the similarities of the input patterns and categorizing (i.e. projecting) them on a (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

3.3.1.8 The Gaussian Layer

Description

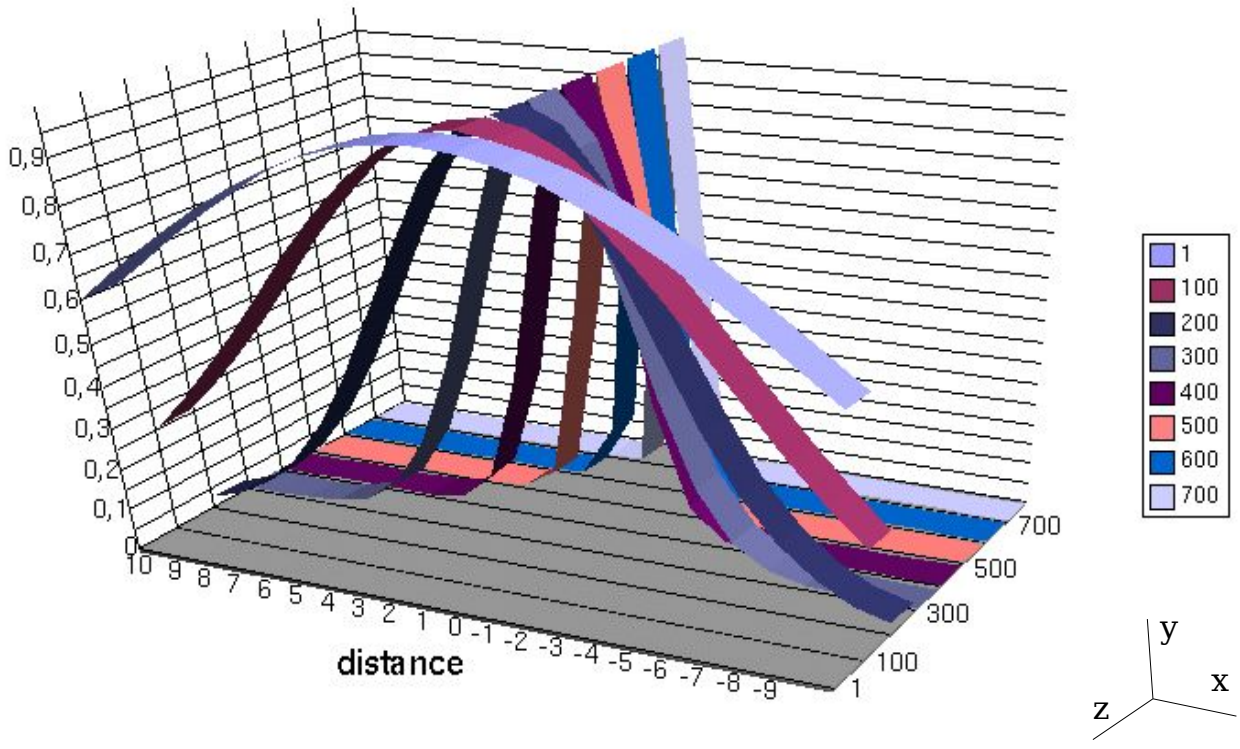
The Gaussian layer performs a similar work like the WTA layer, but in this case it activates the output neurons according a gaussian shape centered around the most active neuron (the winner).
This layer implements the Gaussian Neighborhood SOM strategy. It receives the Euclidean distances between the 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 (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 large the neighborhood size 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

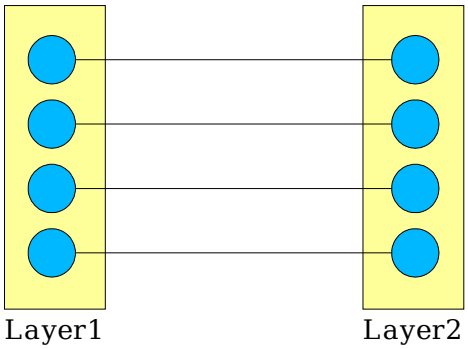
The Synapse represents the connection between two layers, permitting a pattern to be passed from one layer to another.

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 the input or the 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

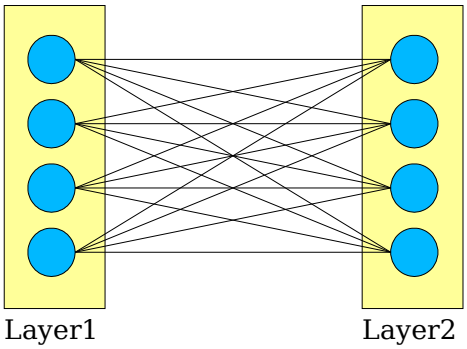
The DirectSynapse represents a direct connection 1-to-1 between the nodes of the two connected layers, as depicted in the following figure:



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:

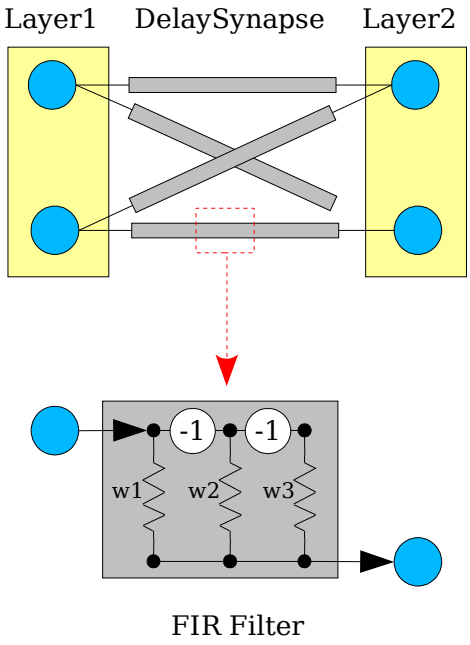


This is the most common type of synapse used in a neural network, and its weights change during the learning phase according to the implemented learning algorithm.
It can connect layers having a whatever number of neurons, and the number of the weights contained is equal to N_1 x N_2, where N_x is the 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.

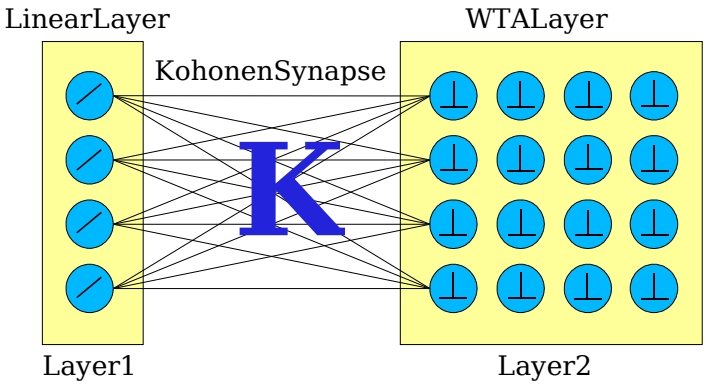
The following figure illustrates how a DelaySynapses can be represented:



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

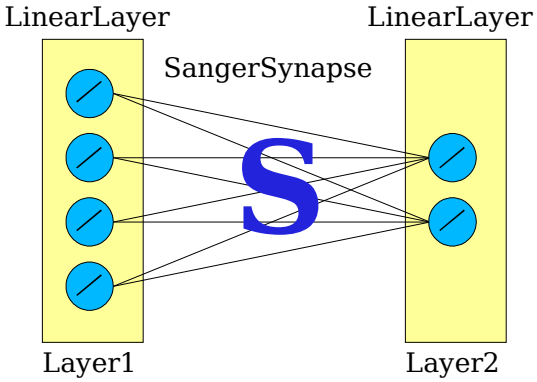
During the training phase, the KohonenSynapse's weights are adjusted to map the N-dimensional input patterns to the 2D map represented by the output synapse.

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' areusing a SOM to recognize handwritten characters, we need a precise response, (like '*the character is A*' ,but 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 unsupervisioned 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:



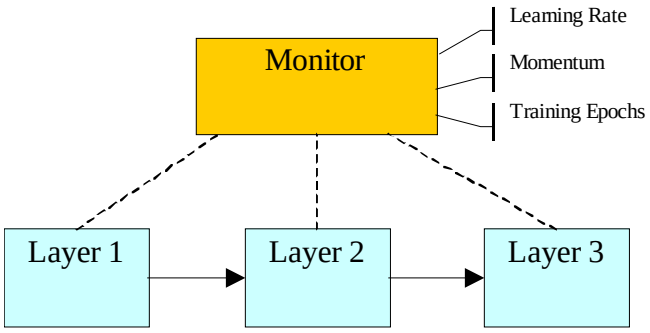
By using this synapse along with the Nested Neural Network component it's very easy to build modular neural networks where the first NN acts as a pre-processing element that reduces the number of the input columns and consequently its noise.

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 a low 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, etc…

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

3.4.1 The Monitor as a container of the NN 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 NN controller

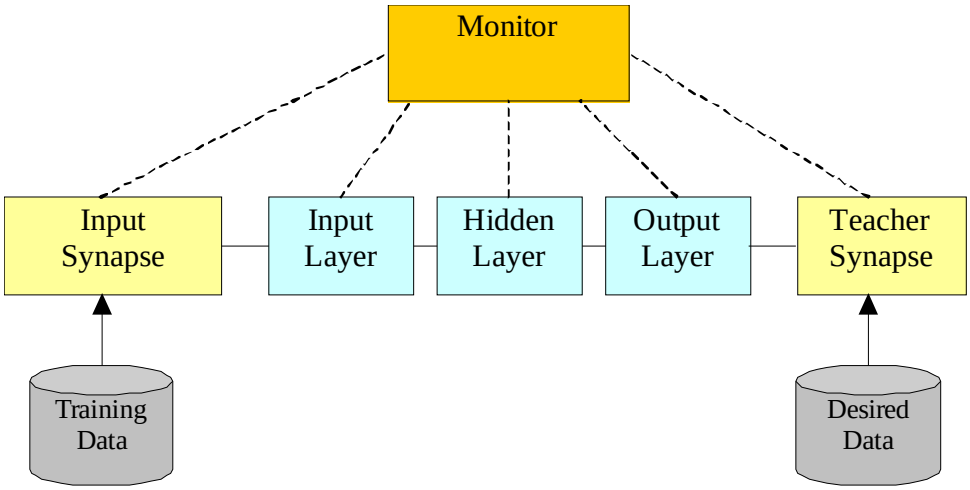
The Monitor object is also a central point for controlling the start/stop times of a neural network.
It has some parameters that are useful to control the behaviour of the NN, e.g. the total number of epochs, the total number of the input patterns, etc.
Before explaining how does this works, an explanation 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 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 (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.

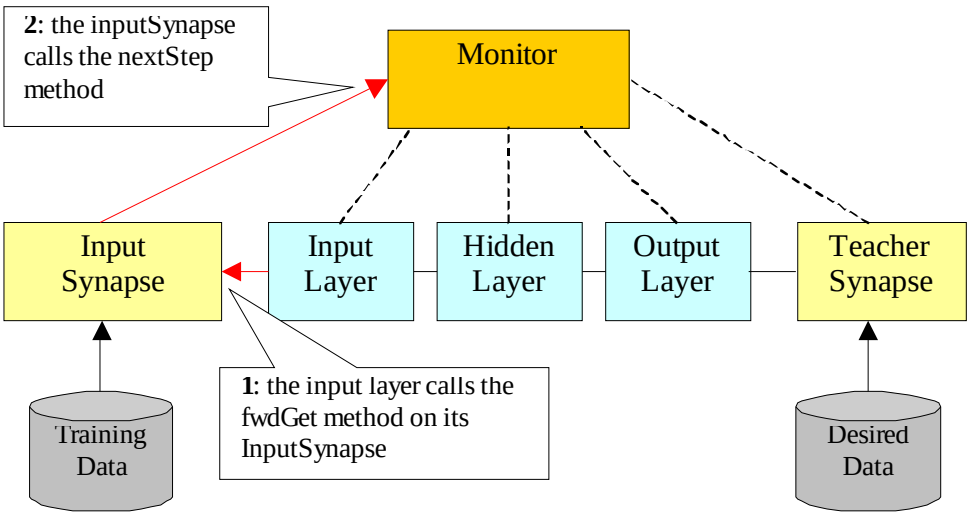
3.4.3 Managing the events

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

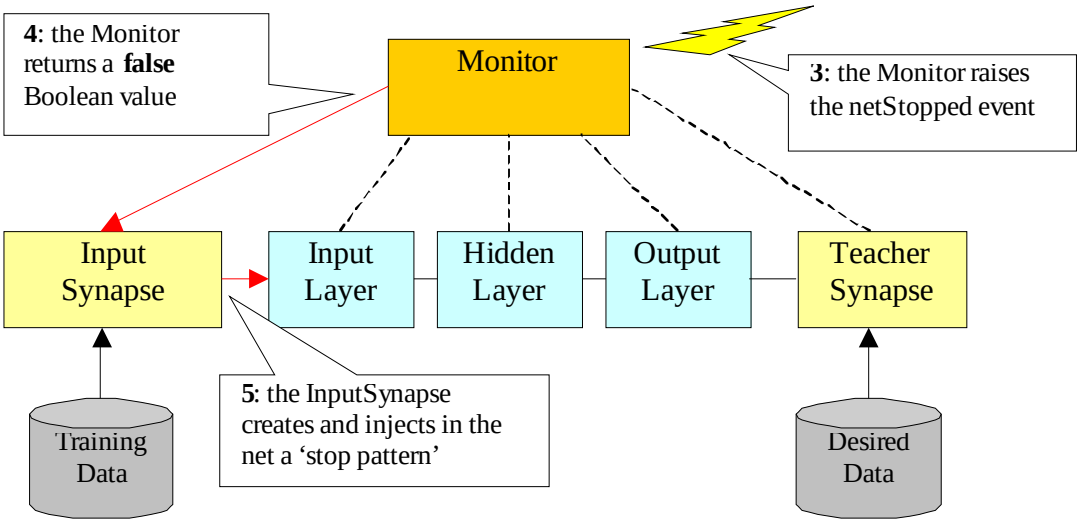


Suppose to have a neural network composed, as depicted in the above figure, of three layers: a InputSynapse to read the training data, a TeacherSynapse to calculate the error for the backprop algorithm, and a Monitor object that controls the overall training process. As already mentioned, all the components 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:

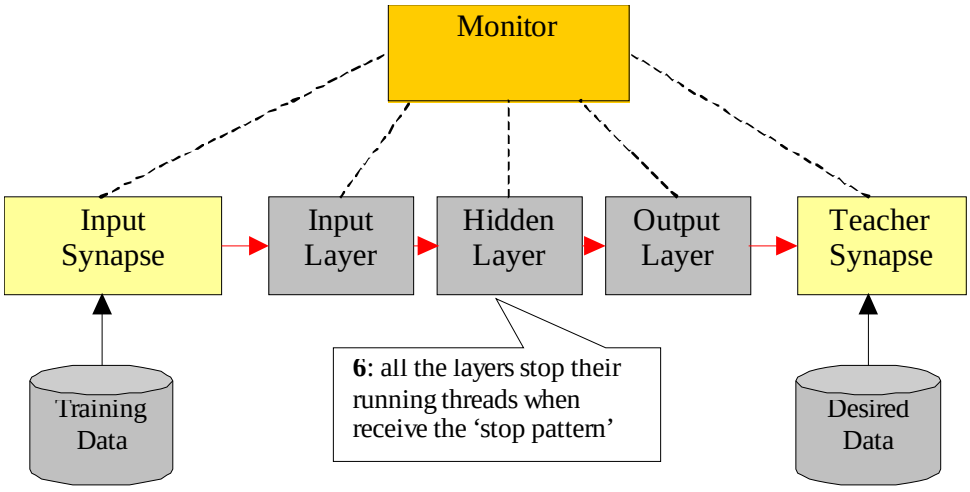


When the input layer calls the InputSynapse (1), the called object interrogates the Monitor to know if the next pattern must be processed (2).



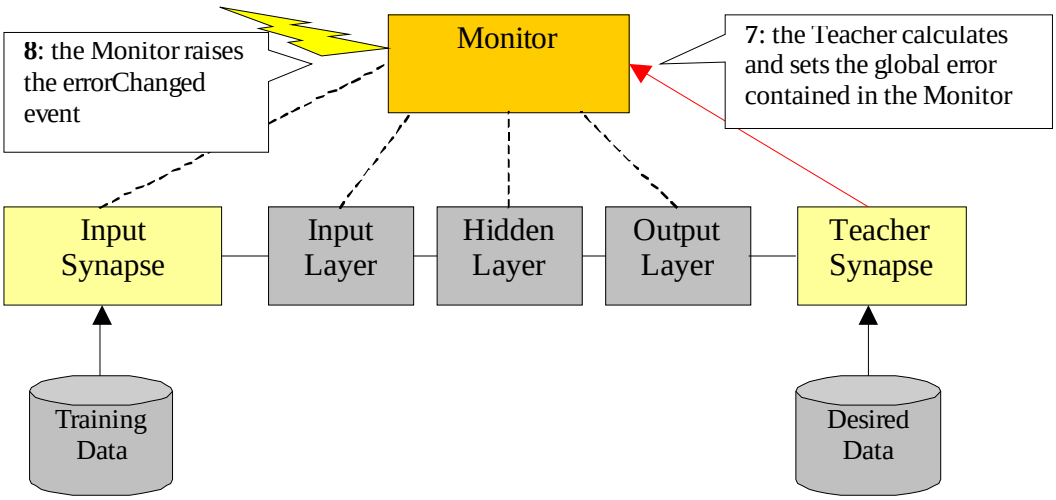
Since, as said, the last epoch is finished, the Monitor object raises a *netStopped* event (3) and returns a **false** Boolean value to the InputSynapse (4).

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.



The TeacherSynapse 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 error: to read the error 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.

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.

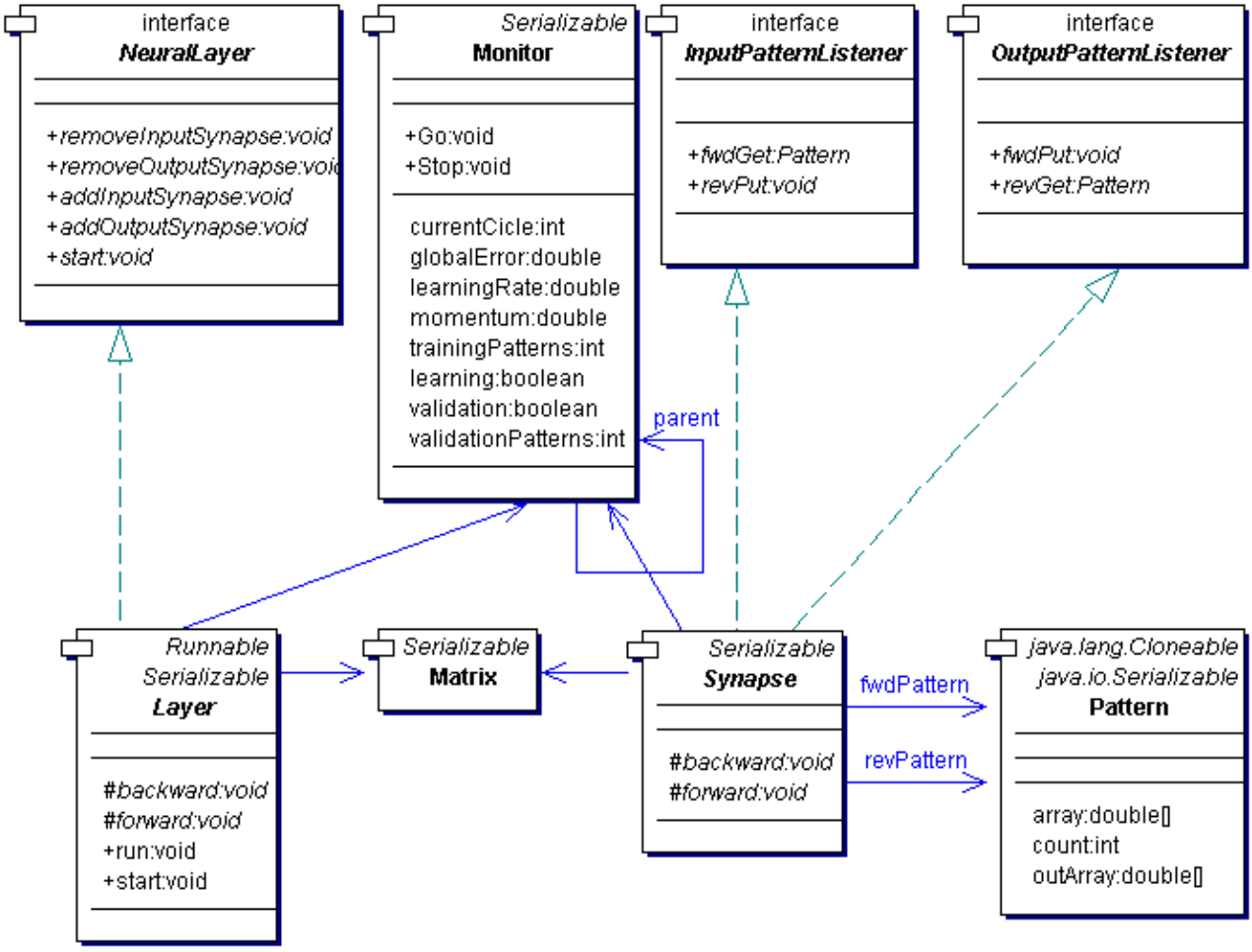
It is composed of two parameters: an array of doubles to contain the values of the transported pattern, and an integer to contain the sequence number of that pattern (the counter).
The dimensions of the array are set according to the dimensions of the pattern transported.
The Pattern object is also used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



To simplify the model, only the relevant properties and methods are shown for each object.
As depicted, 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;
    }
```

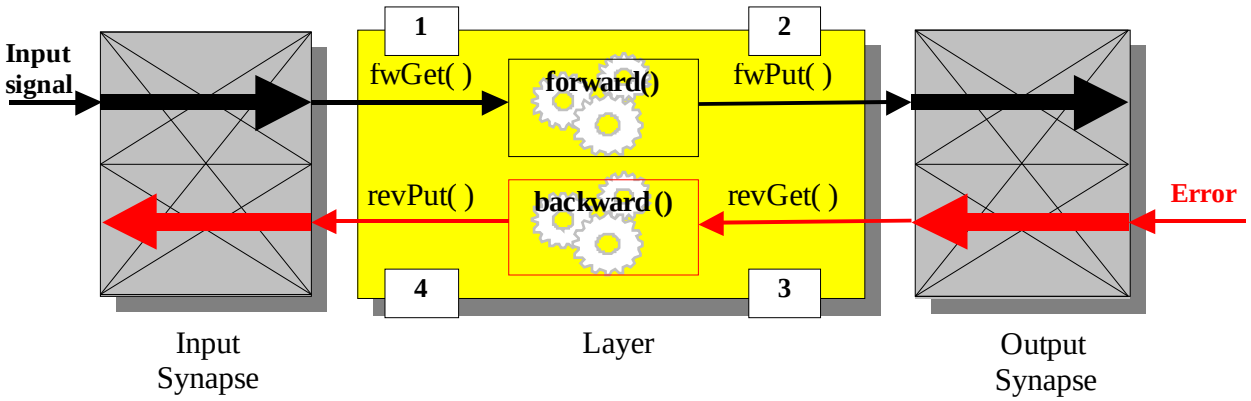
The end of the cycle is controlled by the *running* variable, so the code loops until some ending event occurs.
The two main sections of the code have been highlighted with a border:

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 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 a shared resource of two Layers that, as already mentioned, 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 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.

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

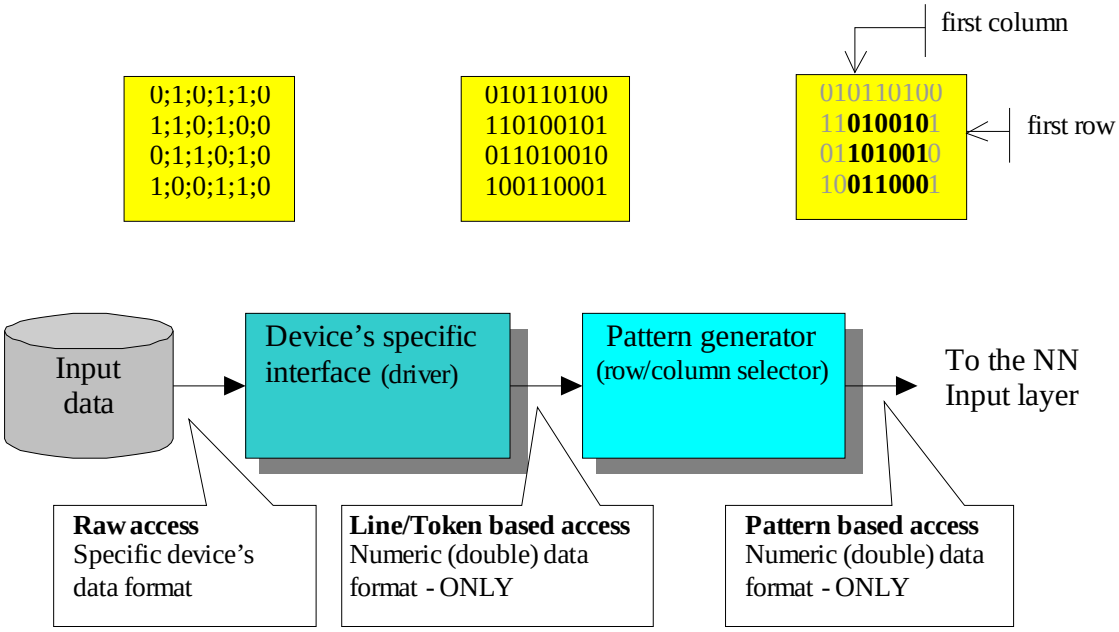
To permit the user to utilize any source of data as input of a neural network, a complete input mechanism has been designed into the core engine.
The main concept underlying the input system is that a neural network elaborates 'patterns'. A pattern is composed by a row of values [x_{11}, x_{12}, …, x_{1N}] 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.
Now 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 overall input system is depicted in the following figure:



Note that the component connected to the first layer of the neural network 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.

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

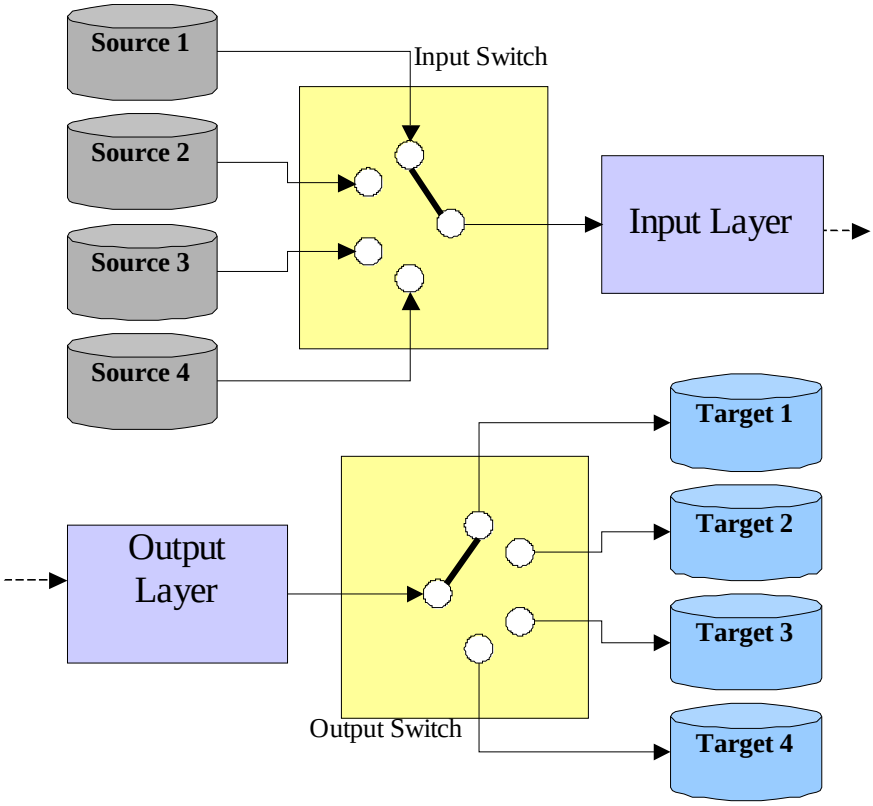
- Java Arrays - to write the output in a 2D array of doubles, to use the output of a neural network from an embedding or external application.

Many others 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

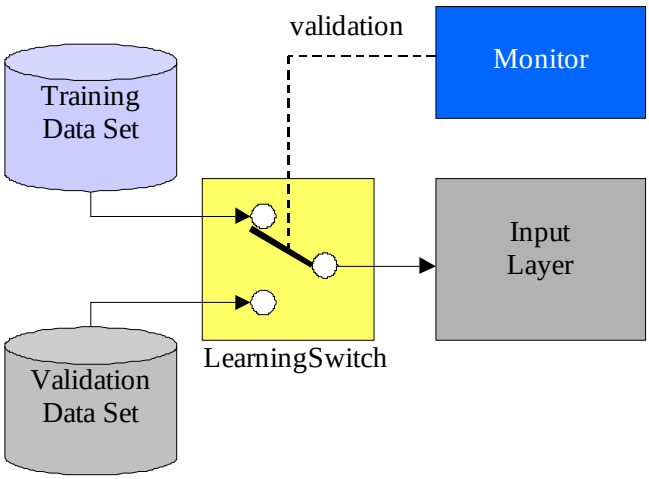
Any Output synapse can be attached to this component. As it acts as an output switch, the active output target can be dynamically changed simply by indicating the name of the output synapse that is to be made the active output of the neural network.

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 them (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.

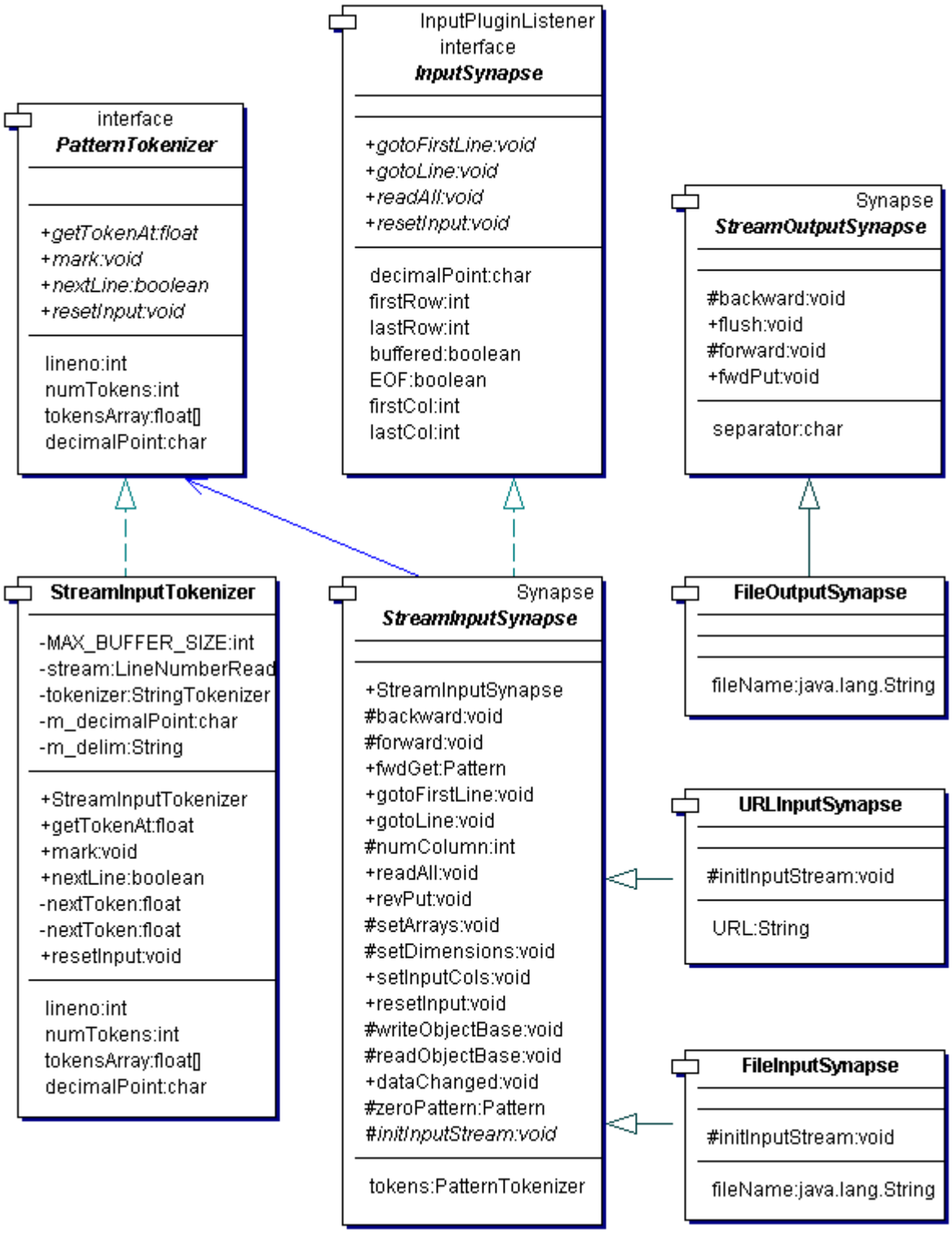
Note that it is not necessary to explicitly set the `validation` parameter of the net before the step 1 because its default value is equal to false (i.e. the training data set is connected to the input layer).

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.

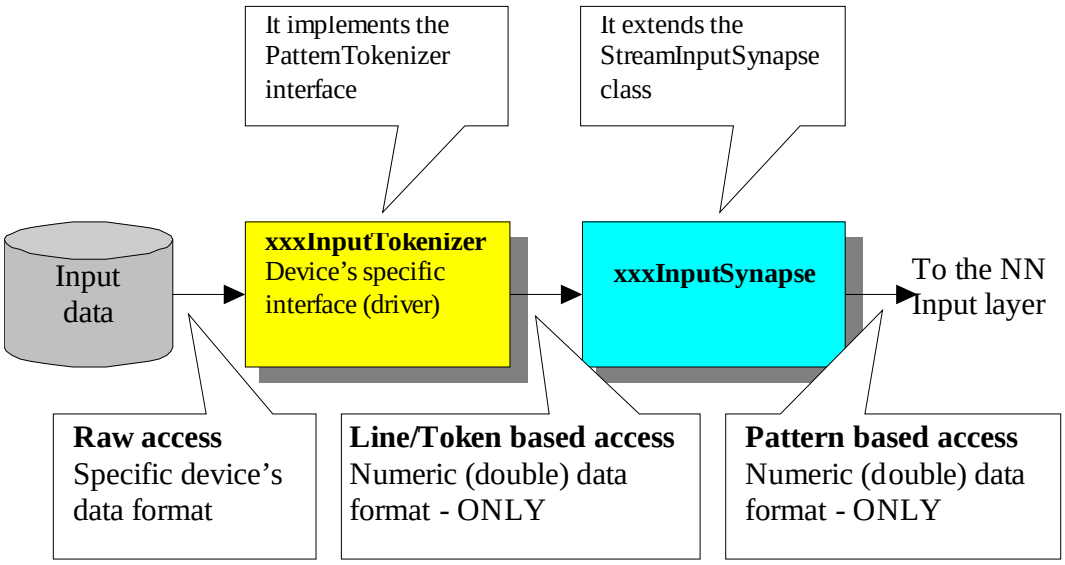
As some files may contain information additional to the required data, the parameters `firstRow`, `lastRow`, `firstCol` and `lastCol`, derived from the `InputSynapse` interface, may be used to define the range of usable data.
The Boolean parameter `stepCounter` indicates if the object is to call the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

By default it is set to TRUE but in some cases it must be set to FALSE. Read below to see why:
In a neural network that is to be trained, there needs to be 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.

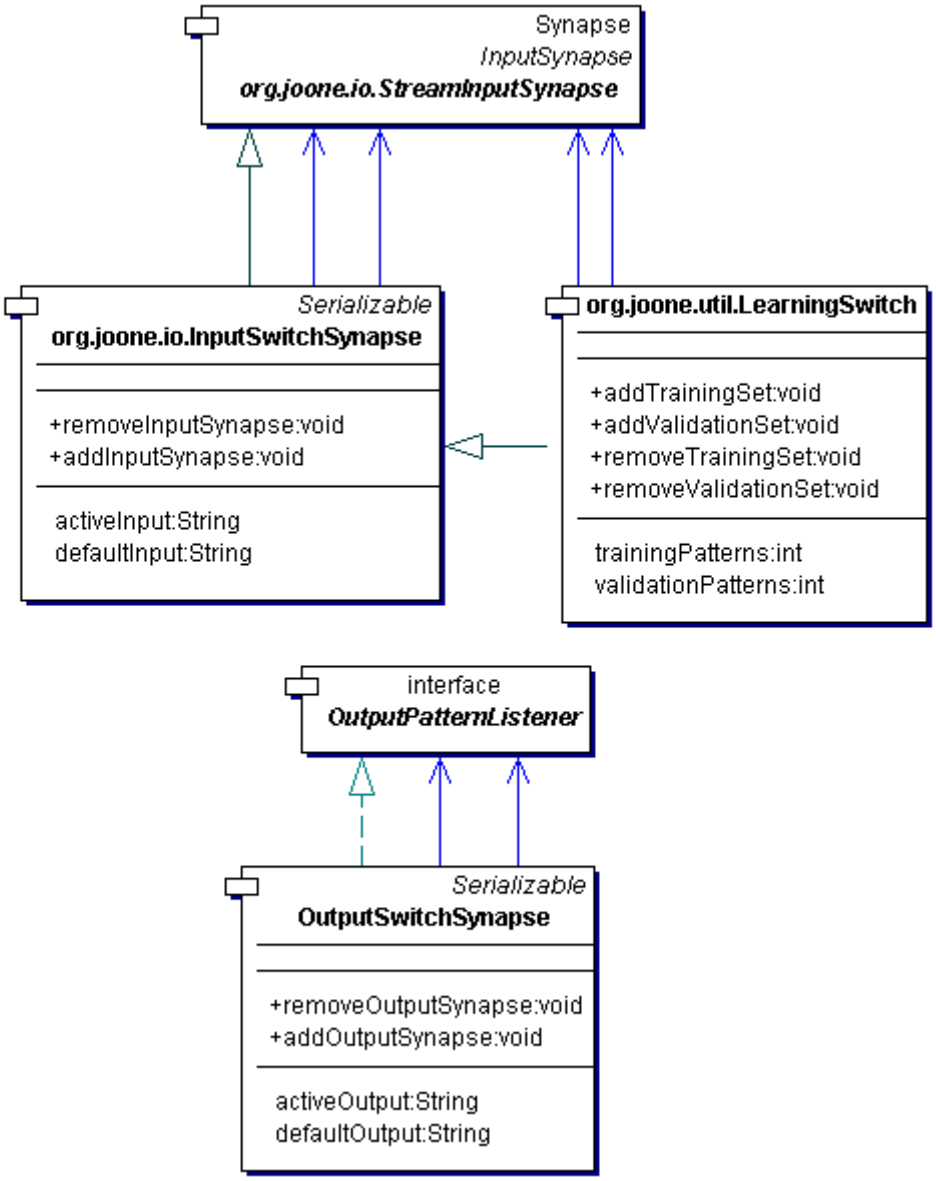
4.5.2 The StreamOutputSynapse

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 is 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 is 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

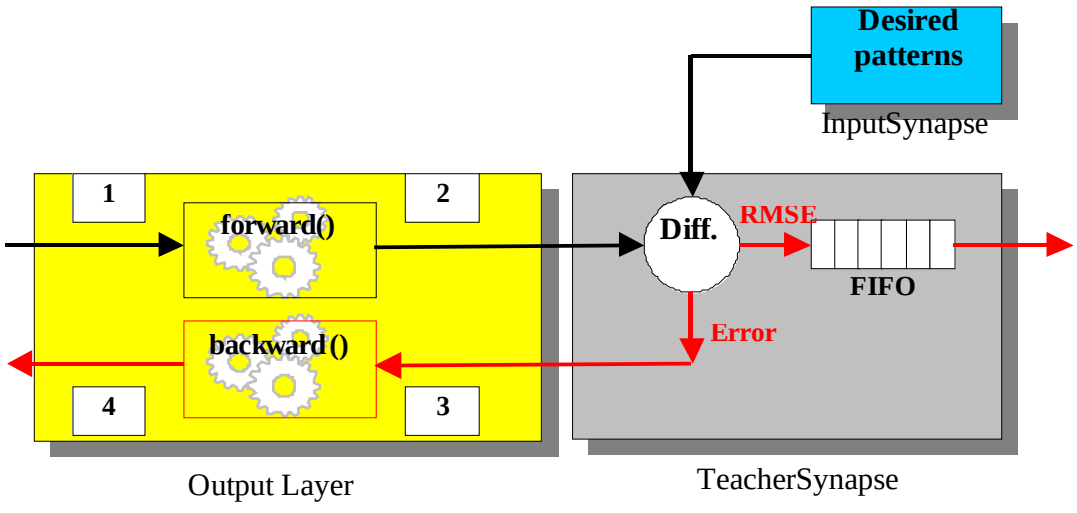
To implement the supervised learning techniques, some mechanism is needed to provide the neural network with the error for each input pattern, expressed as the difference between the output generated by the actual processed pattern and the desired output value for that pattern.

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 specualization.

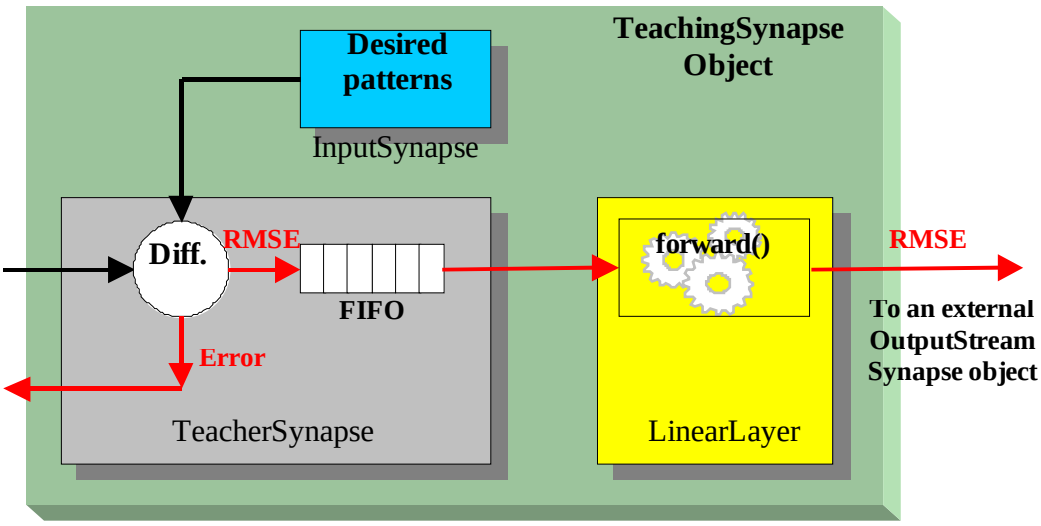
The internal composition of the Teacher object is depicted in the following figure:



The TeacherSynapse object receives – as does any other Synapse – the pattern from the preceding Layer. The Teacher read 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 whatever 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.
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:



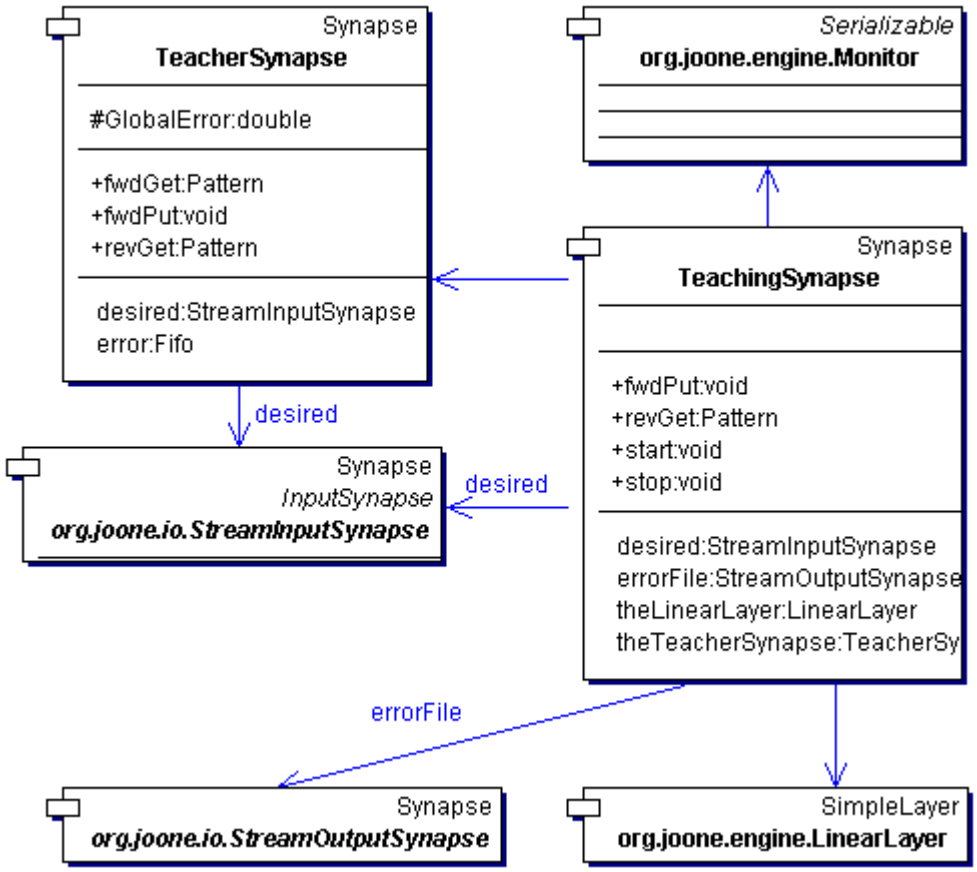
To simplify the construction of the above described chain – `teacher -> fifo -> layer` – a new object (called TeachingSynapse) has been built and inserted in the core engine.

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 Technical details

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, both the TeacherSynapse and the teachingSynapse are represented. The TeachingSynapse is a compound object containing, other that 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 with the StreamInputSynapse 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.

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.
Two main kind of plugins exist in Joone:

1. The input plugins
2. 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 subtracting their average value around the origin
- The **MinMaxExtractorPlugin** to extract the turning points of a time series
- The **MovingAveragePlugin** to calculate the average values of a time series

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

6.2 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:

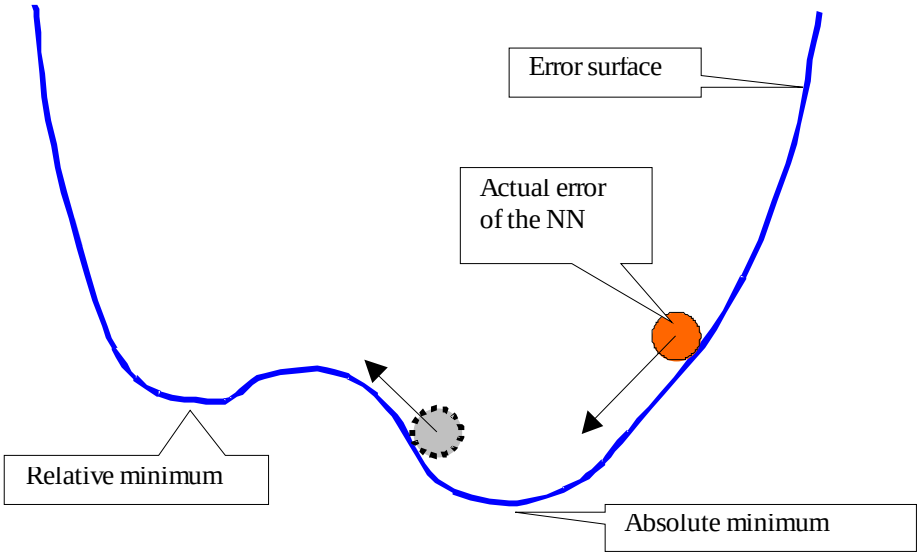
- If E(t) > E(t-1) then LR = LR * (1 - step/100%).
- If E(t) <= E(t-1) then LR remains unchanged.

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 above figure:



All the weights of a network represent an error surface of n-dimensions (for simplicity, in the figure there are only two dimensions). To train 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 velocity is proportionate to the value of the learning rate, so if this velocity is too high, the ball can overstep the absolute minimum and become trapped in a relative minimum.

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

6.3 The Scripting Mechanism

Joone has its own scripting mechanism based on the BeanShell (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.

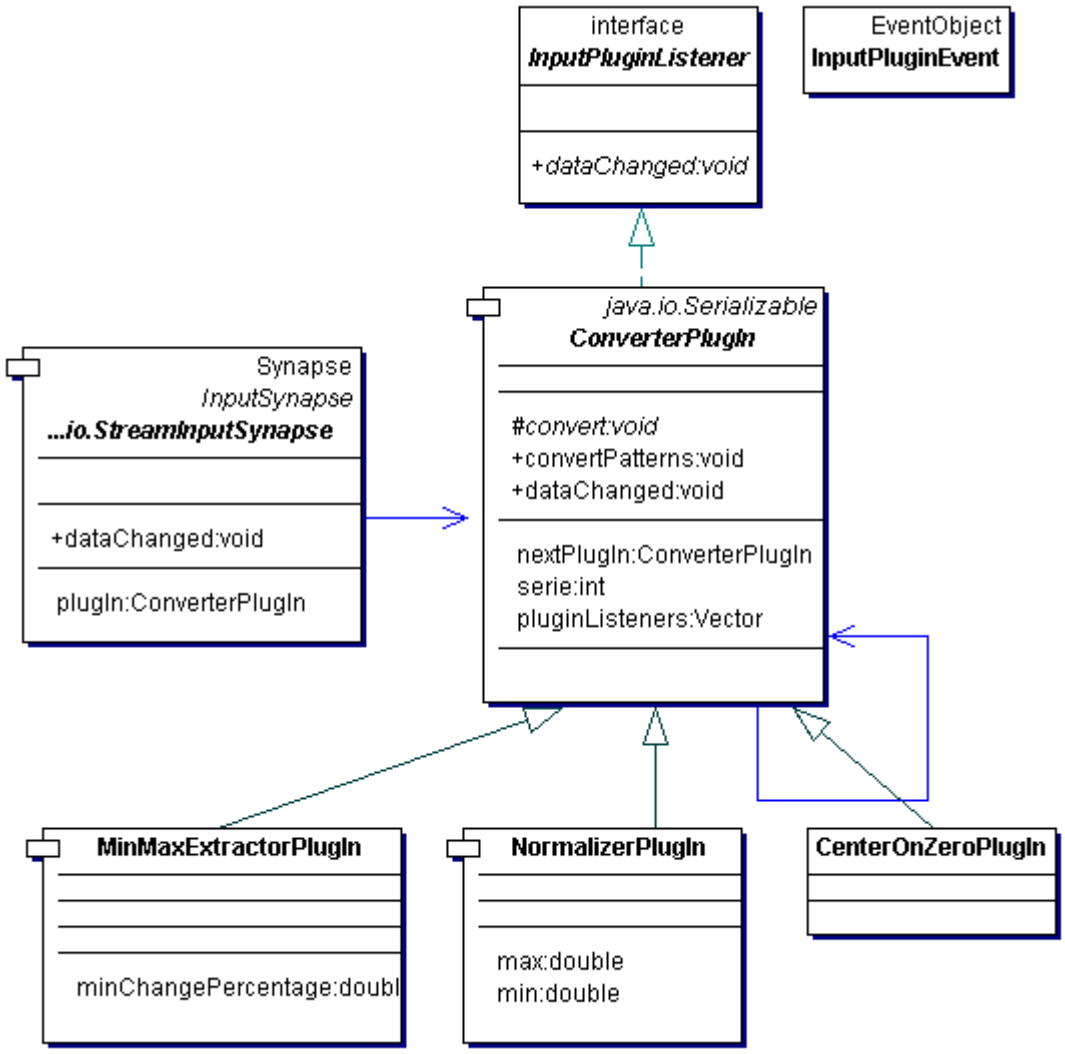
Event-driven macros are all macros associated with the defined events of the neural network. It is possible to execute these scripts in response to a net event. It is impossible to add, remove or rename these macro because they are inherently connected to the events that a neural network can raise. The 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.

6.4 Technical details

6.4.1 The Input Plugins object model

The following figure depicts the object model of the input plugins:



The mechanism, contained in the `org.joone.util` package, is based on the abstract class `ConverterPlugin`.
This 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:

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

Note the methods used to read/write the input values:

- **`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 ConverterPlugin 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 as required. Note the auto-association link on the ConverterPlugin 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.

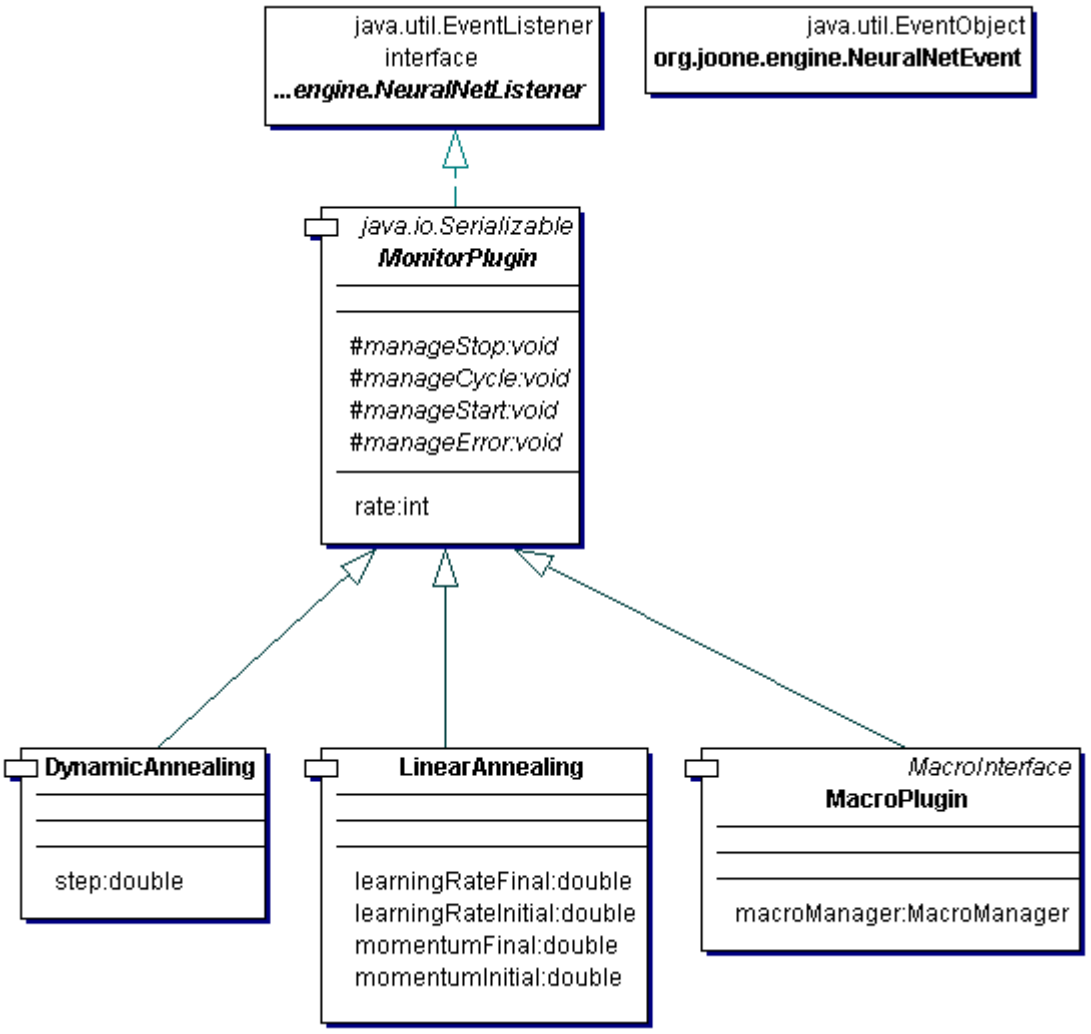
For this reason, when a new input plugin is implemented, the `fireDataChanged` method **must** be called from within the *setXXX( )* method of any parameter that affects the pre-processing calculations.

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();
}
```

6.4.2 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 is written that extends the `MonitorPlugin` class containing the following code:

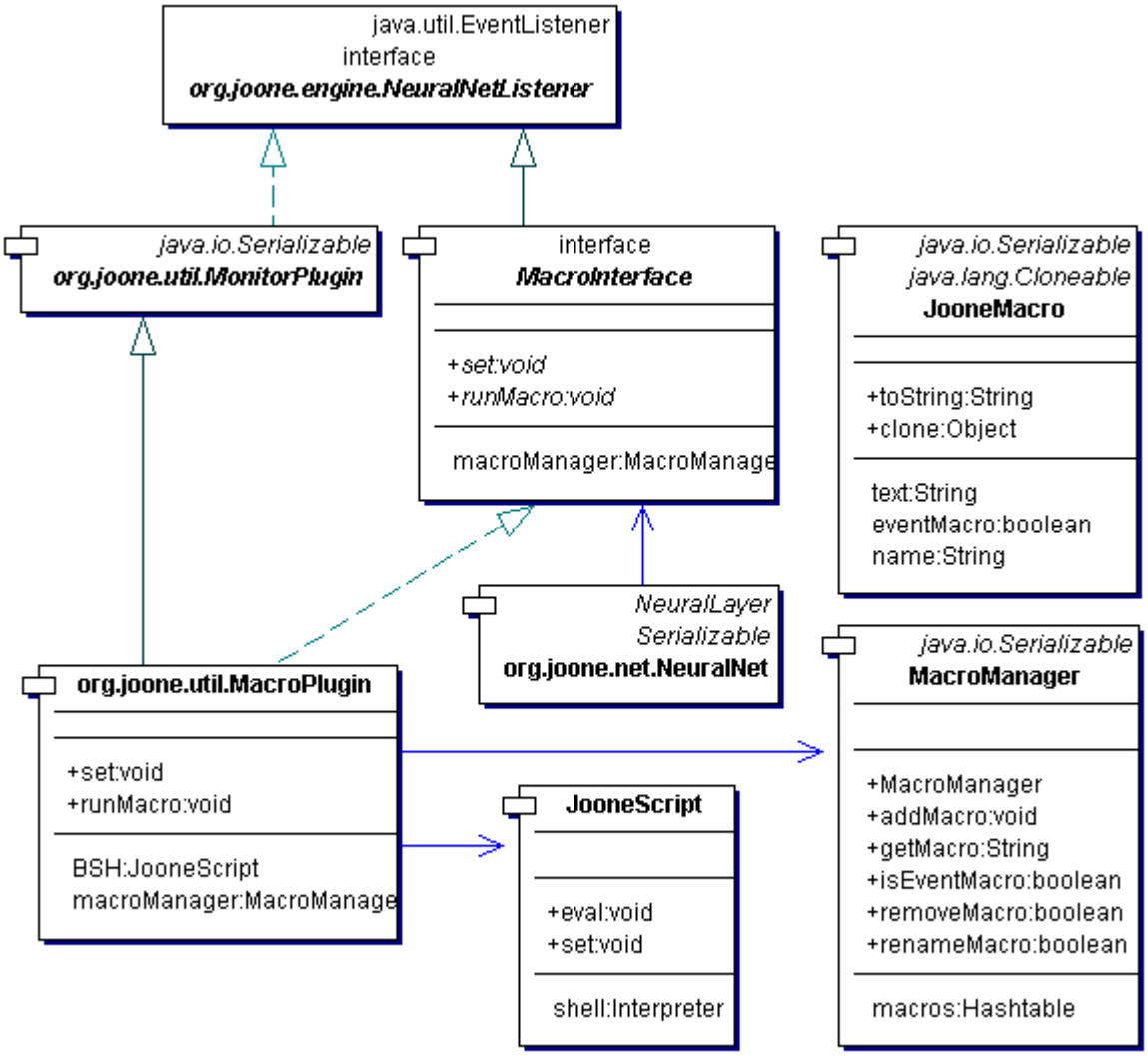
```
Public class stopCondition extends MonitorPlugin {

      protected void manageError(Monitor mon) {
            double rmse = mon.getGlobalError();
            if (rmse < 0.01)
                mon.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.

6.4.3 The Scripting mechanism object model

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 net does not 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.

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.

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. Also it holds a pointer to a TeachingSynapse object and permits this to be externally set.

A neural network 'manager'

The NeuralNet object is also the ‘manager’ of all the behaviour of the contained neural network exposing methods like Start, addNoise, Randomize, resetInput, etc. taking care to apply these methods to its contained components. The Start method, for instance, starts all the Layers of the net, avoiding the user having to invoke this methods on each separate Layer.
The NeuralNet object also provides to the user with some useful features to manage feed forward neural networks.

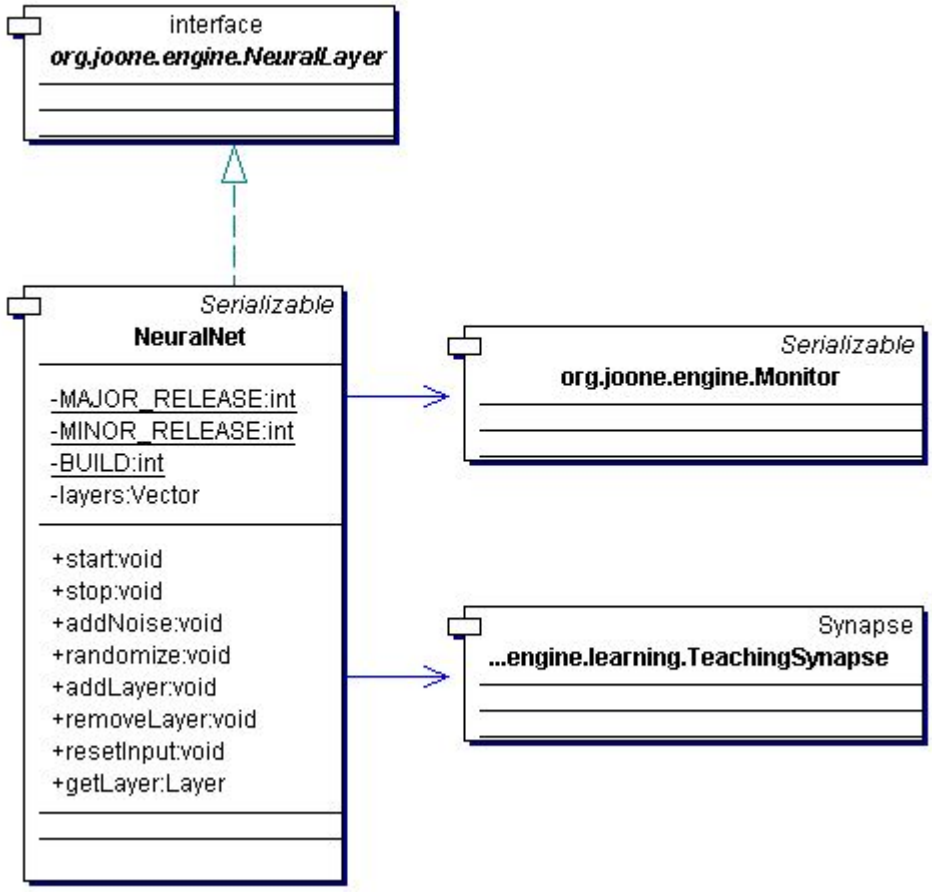
It permits also to manage feed-forward neural network by providing methods to add/remove layers, to get a Layer by its name and to extract the first and the last tier of a neural network, giving either the declared input/output layers, or searching them following these simple rules:

1. A layer is an input layer if:
    a. It has not input synapses connected, or
    b. It has an input synapse belonging to the StreamInputSynapse or the InputSwitchSynapse classes
2. A layer is an output layer if:
    a. It has not output synapses connected, or
    b. It has an output synapse belonging to the StreamOutputSynapse or the OutputSwitchSynapse or the TeacherSynapse or TheachingSynapse classes

The use of these two 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.

7.2 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 engine:



8 Common Architectures

8.1 Temporal Feed Forward Neural Networks

8.1.1 Managing Temporal Series

8.1.1.1 Preprocessing

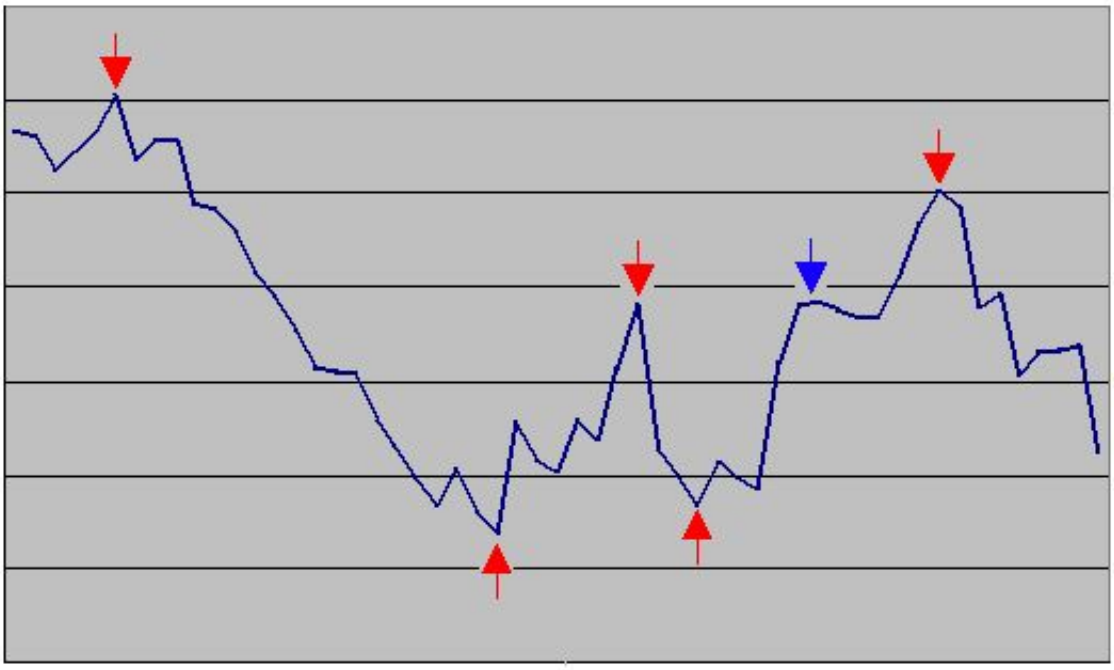
Using a neural network to make financial forecasting is one of the most famous application of the neural networks. In this section we want to explain the use of a pre-processing technique useful to make trend predictions.

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 or short / buy or sell).

Now the next 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:



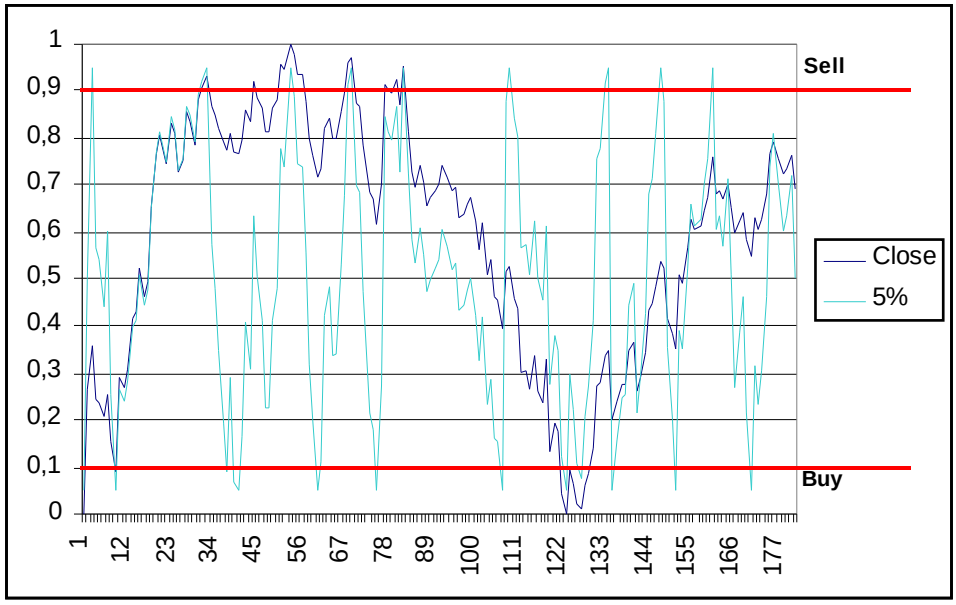
We should trade in correspondence of the red arrows to make money, buying on the lowest values and selling at the higher ones.
A good trading system should raise a signal only when a true turning point is reached, avoiding to generate false signals, as, for instance, that one indicated by the blue arrow, 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).

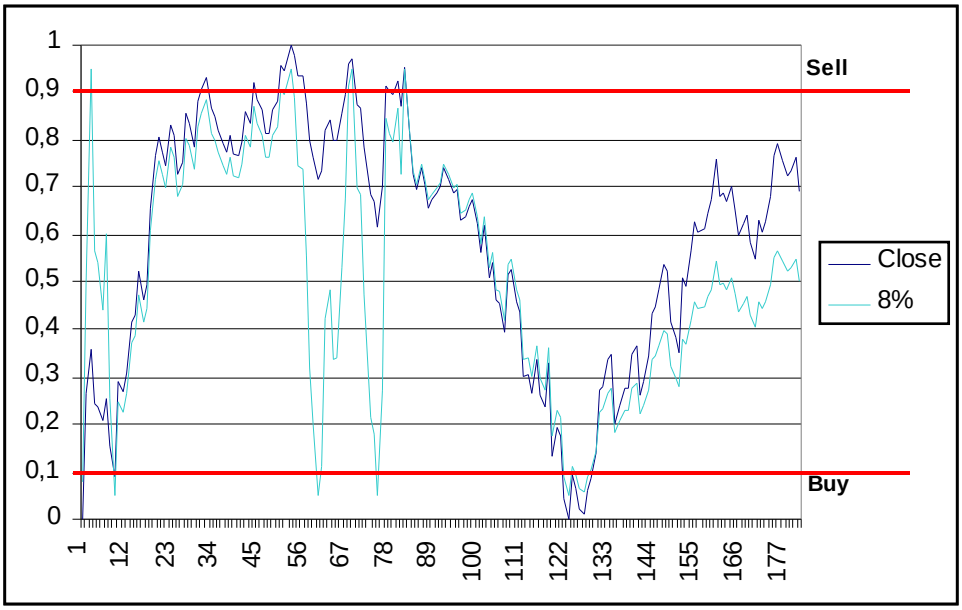
The algorithm is the following:

- 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 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 the signals generated for the 8% case.





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 we can make several experiments to set the above limits and all the other parameters to values that give us the better results.

8.2 Unsupervised Neural Networks

8.2.1 Kohonen Self Organized Maps

8.2.1.1 Example: a character recognition system

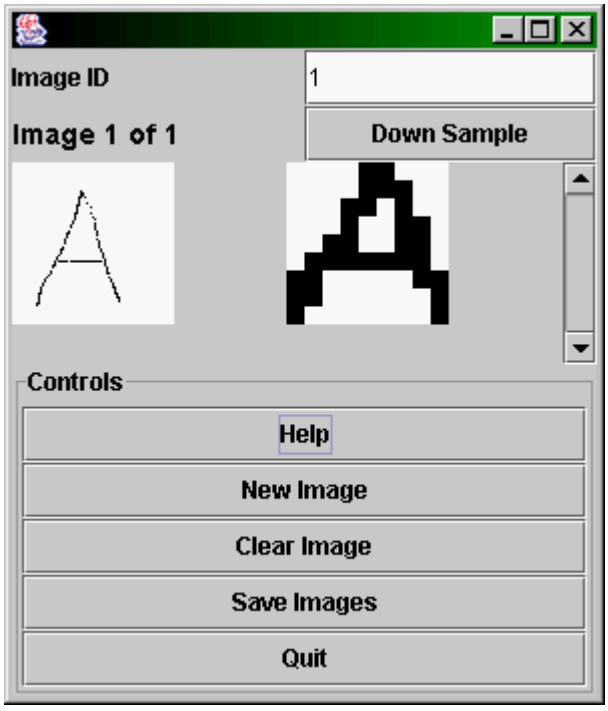
This tutorial is intended to give a basic example of how to perform image / character recognition using SOM / Kohonen neural network architectures.
This tutorial uses a basic application called org.joone.samples.editor.SOMImageTester

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.

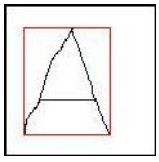


Features

Drawing Area - The high resolution ' Aimage 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.

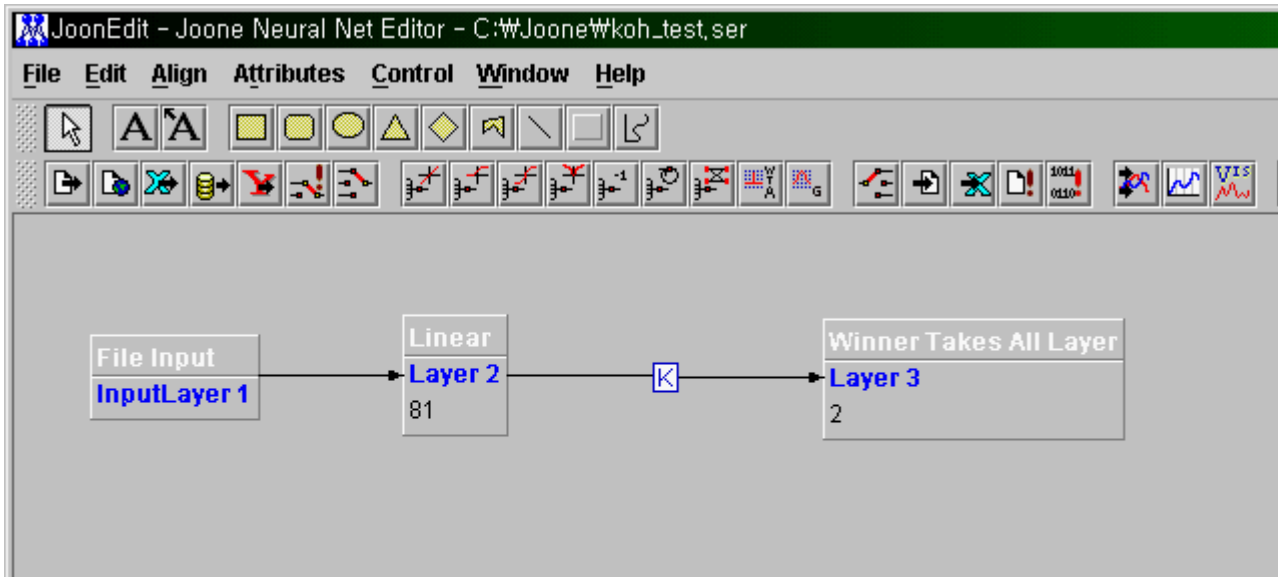
Draw the 4 'A' characters in the drawing panel clicking on New Image when you have finished each one. The down sample button can be used to see what each character looks like down sampled. When you have finished drawing the 4 'A' characters then draw four 'B' characters. Then use the Save Images button 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.

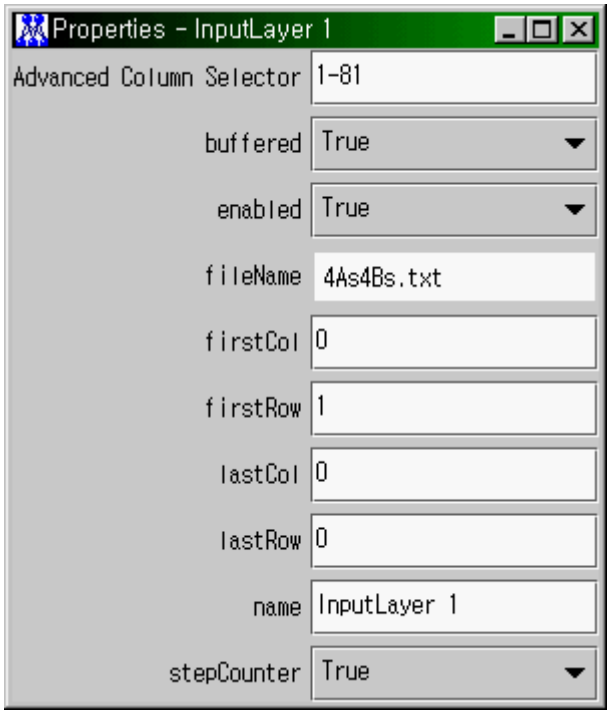
Neural Network Setup

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 in. See the image below ...



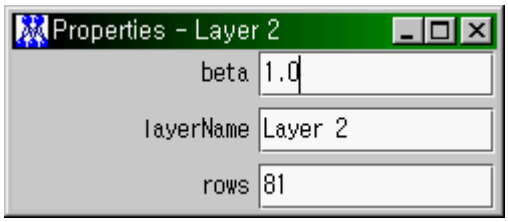
Note the Winner Takes All layer has two neurons, this is to ensure it classifies out two characters.

Input Layer Properties



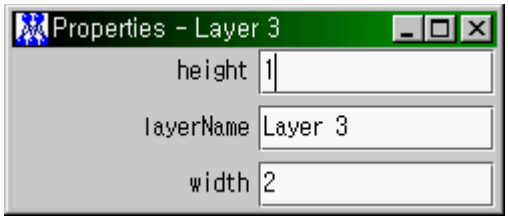
Note our input images have 81 inputs i.e the 9x9 down sampled image that the application made earlier.

Linear Layer Properties



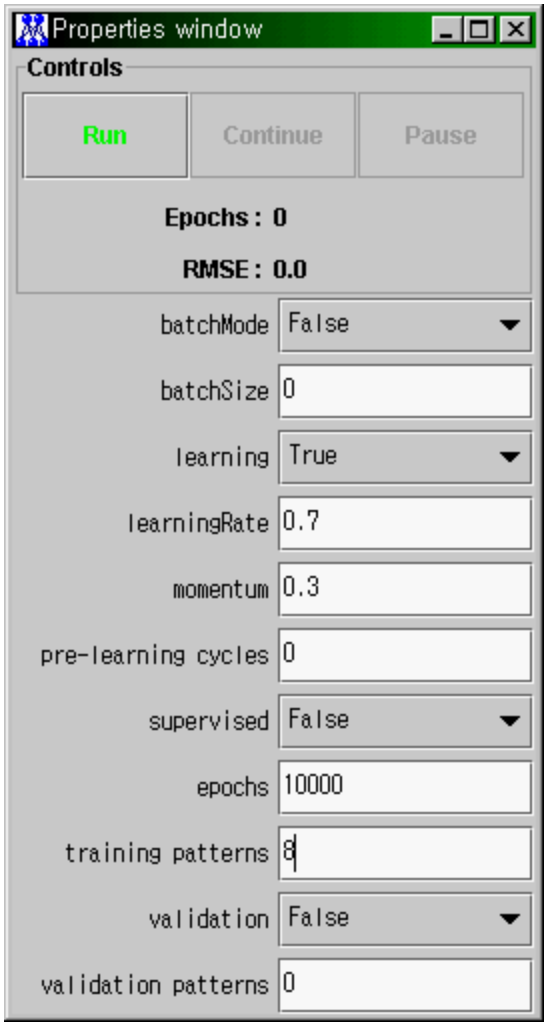
Note the rows here must match the inputs from the file input synapse.

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 syanpse , 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. At the time of writing the Joone project is looking the producing an Image Input Synapse that will enable users to present images from files or Java images. If this is available then you could use this to easily load images into the network for training and running.
If this is not available then you will have to write some image pre-processing in your custom application.

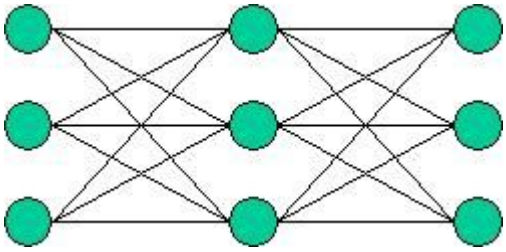
Something worth thinking about when looking at image recognition is things like colour , size , shape, texture etc. An extension to the this example might be to enable the net to recognise coloured characters but independent of the actual colour. If you always present 'A' in green and 'B' in blue and 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.

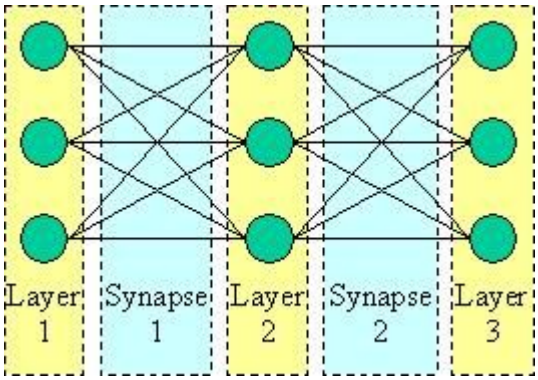
9 Applying Joone

9.1 A simple (but useless) neural network

Consider a feed-forward neural net 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 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

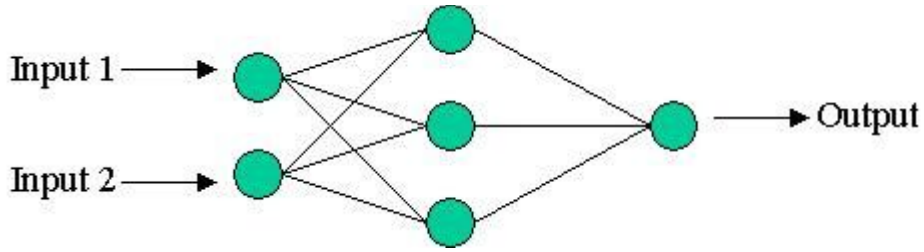
Firstly, a file containing these values is created:

```
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 SygmoidLayer();
```

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 create a Monitor object to provide the net with all the parameters needed for it to work:

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

Give the layers a reference to that Monitor:

```
input.setMonitor(monitor);
hidden.setMonitor(monitor);
output.setMonitor(monitor);
```

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.setFileName("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.setFileName("c:\\joone\\XOR.txt");
trainer.setDesired(samples);
/* The output values are on the third column of the file */
samples.setAdvancedColumnSelector(“3”);
/* We give it the monitor's reference */
trainer.setMonitor(monitor);
```

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

```
output.addOutputSynapse(trainer);
```

Now all the layers must be activated by invoking their method start. The layers implement the java.lang.Runnable interface, in that way they run on separated threads.

```
input.start();
hidden.start();
output.start();
```

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 */
monitor.Go(); /* The net 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");
        System.exit(0);
    }

    public void cicleTerminated(NeuralNetEvent e) {
        Monitor mon = (Monitor)e.getSource();
        long c = mon.getCurrentCicle();
        long cl = c / 1000;
        /* We want print the results every 1000 cycles */
        if ((cl * 1000) == c)
            System.out.println(c + " cycles remaining - Error = " +
mon.getGlobalError());
    }
```

(The source code can be found in the CVS repository in the org.joone.samples.engine.xor package)

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.

9.3.1 The simplest way

A simple way to save a neural network is to serialize each layer 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(input);
     out.writeObject(hidden);
     out.writeObject(output);
     out.writeObject(trainer);
     out.close();
    }
catch (Exception excp) {
      excp.printStackTrace();
      }
}
```

We don't need to explicitly save the synapses constituting the neural network, because they are linked by the layers. 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 void restoreNeuralNet(String filename) {
try {
     FileInputStream stream = new FileInputStream(fileName);
     ObjectInputStream inp = new ObjectInputStream(stream);
     Layer input = (Layer)inp.readObject();
     Layer hidden = (Layer)inp.readObject();
     Layer output = (Layer)inp.readObject();
     TeachingSynapse trainer = (TeachingSynapse)inp.readObject();
    }
catch (Exception excp) {
     excp.printStackTrace();
     }
```

```
        /*
         * After that, we can restore all the internal variables to manage
         * the neural network and, finally, we can run it.
         */

        /* We restore the monitor of the NN.
         * It's indifferent which layer we use to do this */
        Monitor monitor = input.getMonitor();
        /* The main application registers itself as a NN's listener */
        monitor.addNeuralNetListener(this);
        /* Now we can run the restored net */
        input.start();
        hidden.start();
        output.start();
        monitor.Go();
}
```

The method illustrated in this chapter is very simple and works well, but it's not flexible enough, because we have to write a different piece of code for each saved neural network, as the number and the order of the saved layers of the network is hard-coded in the program.
We now consider a quicker and more flexible method to save and restore a neural network.

9.3.2 Using a NeuralNet object

The org.joone.net.Neuralnet object comes in our aid by offering a simple but powerful mechanism to manage a neural network built with Joone.
We now will try to rewrite the XOR sample using this new component.

In any case we must create all the necessary components of the neural network, repeating all the instructions already written for the previous example:

```
        /* The Layers */
        LinearLayer input = new LinearLayer();
        SigmoidLayer hidden = new SigmoidLayer();
        SigmoidLayer output = new SygmoidLayer();
        input.setRows(2);
        hidden.setRows(3);
        output.setRows(1);

        /* The Synapses */
        FullSynapse synapse_IH = new FullSynapse(); /* Input -> Hidden conn. */
        FullSynapse synapse_HO = new FullSynapse(); /* Hidden -> Output conn. */
        input.addOutputSynapse(synapse_IH);
        hidden.addInputSynapse(synapse_IH);
        hidden.addOutputSynapse(synapse_HO);
        output.addInputSynapse(synapse_HO);

        /* The I/O components */
```

```
        FileInputSynapse inputStream = new FileInputSynapse();
        inputStream.setAdvancedColumnSelector(“1,2”);
        inputStream.setFileName("c:\\joone\\XOR.txt");
        input.addInputSynapse(inputStream);

        /* The Trainer and its desired file */
        TeachingSynapse trainer = new TeachingSynapse();
        FileInputSynapse samples = new FileInputSynapse();
        samples.setFileName("c:\\joone\\XOR.txt");
        trainer.setDesired(samples);
        samples.setAdvancedColumnSelector(“3”);
        output.addOutputSynapse(trainer);
```

Now we add this structure to a NeuralNet object:

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

and we instead use the contained Monitor object to create a new one:

```
        Monitor monitor = nnet.getMonitor();
        monitor.setLearningRate(0.8);
        monitor.setMomentum(0.3);
        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 */
              monitor.addNeuralNetListener(this);
```

and now we can run the neural network simply writing:

```
        nnet.start();
        nnet.GetMonitor().Go();
```

Where are the differences?

1. We don’t need any more to set the Monitor object for each component, as the NeuralNet does this task for us;
2. We don’t need to invoke the start method for all the layers, but only on the NeuralNet object.

But the main support provided by the NeuralNet object is the ability to easily store and read a neural network with few and generalized rows of code:

```
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();
      }
```

```
}

public void restoreNeuralNet(String filename) {
try {
     FileInputStream stream = new FileInputStream(fileName);
     ObjectInputStream inp = new ObjectInputStream(stream);
     nnet = (NeuralNet)inp.readObject();
    }
catch (Exception excp) {
      excp.printStackTrace();
     return;
      }
     /*
      * After that, we can restore all the internal variables to manage
      * the neural network and, finally, we can run it.
      */

     /* The main application registers itself as a NN's listener */
     nnet.getMonitor().addNeuralNetListener(this);
     /* Now we can run the restored net */
     nnet.start();
     nnet.getMonitor().Go();
}
```

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) {
      NeuralNet nnet = null;
      try {
            FileInputStream stream = new FileInputStream(fileName);
            ObjectInputStream inp = new ObjectInputStream(stream);
            nnet = (NeuralNet)inp.readObject();
      }
      catch (Exception excp) {
            excp.printStackTrace();
      }
      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 for only one cycle in recall mode
      xorNNet.getMonitor().setTotCicles = 1;
      xorNNet.getMonitor().setLearning(false);
      xorNNet.start();
      xorNNet.getMonitor().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 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.

9.4.3 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.start();
        xor.getMonitor().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 *synchronized* 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) {
        if (args.length < 1) {
            System.out.println("Usage: EmbeddedXOR XOR.snet");
        }
        else {
            EmbeddedXOR xor = new EmbeddedXOR();
            xor.Go(args[0]);
        }
    }
```

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

9.4.4 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);
            ...
```

As you can read, we now use as input 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 Technical Overview 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), while during the training phase the start and stop actions was determined by the parameters of the Monitor object, being that process not supervisioned (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 call the Monitor.Go method, nor to set its 'TotCycles' and 'Patterns' parameters.

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

```
            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
                double[] pattern = memOut.getNextPattern();
                System.out.println("Output Pattern #"+(i+1)+" = "+pattern[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.

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!