TUTORIAL
NEUROPH – AN OPEN SOURCE DEVELOPMENT PLATFORM FOR NEURAL NETWORKS

Zoran Sevarac [1], Vladan Devedzic[2]

Abstract:
Neuroph is an open source software that provides Java-based software components and tools for developing neural networks. It is designed to be easy to learn, use, extend and customize for specific needs. Thanks to these features, Neuroph has gained wide adoption in the world-wide user community. This tutorial explains details about the Neuroph platform design and demonstrates its usage through examples, both in Java code and with tools with graphical user interface. The examples include an introductory (educational) classification demo with visualization, a basic classification demo with complete work-flow for neural network training and cross-validation, and a basic image recognition demo. It also demonstrates how it can be extended and used in specific application domains. The Neuroph platform is based on best software engineering practices, experience from other neural network software developed in Java, and on proven Java software industry standards like NetBeans Platform.

Keywords: neural networks, software, framework, Java

1. Introduction

Neuroph[3] is an open source software platform for neural network development, education and research. It provides a well-designed software development framework written in Java programming language [1], and an integrated development environment for neural networks, with graphical user interface (GUI) based on NetBeans Platform [2]

Neuroph enables and makes easier the development of various types of neural networks, and their integration into the real world applications. It is primarily intended to be used by Java software developers and students. However, thanks to its flexibility and openness, it is also an interesting option for neural network research.

Although there are many similar software frameworks and tools for neural networks developed in Java over the last decade [3] [4] [5] [6] [7] [8] [9] [10] [11], Neuroph managed to gain popularity and build community thanks to its ease of use and extensibility. Neuroph makes easy for software developers to learn about neural networks, create software components that use neural networks, and extend or customize existing types of neural networks (which are provided by the underlying software framework).

Neuroph is being developed at the University of Belgrade with contributions from individual developers and universities all around the world [12].It has stable development history since its first release in 2008,with one release per year. At the moment, it has over 900 downloads per week,

[1] Faculty of Organizational Sciences, University of Belgrade, Jove Ilica 154, Belgrade, E-mail: sevarac.zoran@fon.bg.ac.rs
[2] Faculty of Organizational Sciences, University of Belgrade, Jove Ilica 154, Belgrade, E-mail: devedzic.vladan@fon.bg.ac.rs
[3] http://neuroph.sourceforge.net

according to the statistics from the official project download page[4].

It is being used for teaching neural networks at the University of Belgrade in the course on Intelligent Systems[5], and it has won the Duke's Choice Award 2013[6], which is the annual Java community award for the most innovative software on Java platform.

Latest version of Neuroph software v2.92 is available for download in executable form, with full source code and documentation from official Neuroph site[7]

This paper describes design, tools and main features of the Neuroph platform, through three application examples. It explains how use Neuroph to solve particular tasks, and provides information to understand Neuroph in order to be able to extend it or customize it. The paper is organized as follows: Section 2 provides an overview of Neuroph's internal design and tools. Section 3 provides a detailed description of three application cases, which demonstrate the usage and the most important features of the Neuroph software. These application cases include: an animated educational classification example, a simple Iris flower classification example, and an image recognition example. Section 4 gives an overview of the main extension points and guidelines for extending and customizing Neuroph. Some concluding remarks are given in section 5.

2. Neuroph Design and Tools

This section describes the general design, components and tools of the Neuroph platform. The main components of the Neuroph platform are:

1. Neuroph framework, which provides a Java class library for various neural network components; and
2. GUI tool called Neuroph Studio, which provides integrated development environment for neural networks based on the Neuroph framework and NetBeans Platform.

2.1. Design of Neuroph Framework

Neuroph is designed with the goal to provide easy to use, flexible and extensible environment for creating neural networks and making them part of a tool-set of Java software developers. So the main objectives of Neuroph design are to maximize the following factors of software quality:

1. Usability – Neuroph should be easy to learn and use. In practice this means that there should be a small number of classes and methods required to learn, along with tools that make it easier to create, simulate, and analyze neural networks.

2. Flexibility and extensibility – it should be easy to extend Neuroph (add new features) and customize it (modify existing features) for specific needs.

3. Re-usability – it should be easy to deploy Neuroph in various environments and application domains, with high level of code reuse when developing extensions based on existing Neuroph components.

[4] http://sourceforge.net/projects/neuroph/files/
[5] http://ai.fon.bg.ac.rs/osnovne/inteligentni-sistemi/
[6] http://neuroph.sourceforge.net/dukes_choice_award_2013.html
[7] http://neuroph.sourceforge.net/download.html

In order to meet these goals, domain-driven and hot-spot-driven design principles were applied.
Domain driven approach [13]means to base a complex design on a domain model, which results in better understandability and learn-ability. In the case of the neural network framework design, this means choosing framework elements so that they correspond to neural network domain concepts, which makes the design easier to understand.
Following these principles, the main concepts in the neural network domain used for Neuroph design are:

- Neural Network – a network that consists of a *layers* of interconnected processing units (*neurons*) that can learn from some *data set* using a corresponding *learning algorithm*.
- Learning algorithm – an algorithm that adjusts the network parameters (connection weights) during the training procedure until the network gets a desired behavior
- Data Set – a collection of data that is used in the training procedure and for testing the network
- Layer – a collection of neurons
- Neuron – a basic processing unit in the network; can use various input and transfer functions
- Connection – connection between two neurons with associated Weight

Hot-spot-driven design [14] [15]is about aspects that have to be kept flexible using appropriate abstractions, so these can be easily changed. The problem with framework design is how to balance the flexibility versus the complexity of use of that framework [4]. The complexity of keeping the framework exceedingly flexible also makes the framework harder to reuse and maintain. Less flexible solutions tend to keep the framework smaller and easier to maintain and reuse. Thus to find out the most important hot spots becomes a major issue when designing a framework [16].
The most general requirement for flexibility of a neural network framework is to support different types of neural network architectures and learning algorithms, and make them domain/application independent. In order to achieve that, the design needs to support different types of neurons, input and transfer functions, connectivity patterns and interaction between neurons, and different types of learning algorithms. All of these represent 'hot spots' of the framework.
The goal of Neuroph's design regarding flexibility was to come with generic base design that can be extended to create any possible type of neural network, or at least to create the design that is possible to evolve into that direction over time.
Regarding re-usability, the design objectives are to enable creating new types of neural networks, learning algorithms, and other neural network components, with minimal coding effort, and to allow reuse of existing components. Also, the design should support reuse of neural network components in different applications, as well as use of different types of neural networks in the same application with minimal change. In addition, it should support commonly adopted procedures for neural network training and development regardless of the application domain.
Following these ideas and guidelines, the core design of the Neuroph framework has evolved as shown on the class diagram in Figure 1.

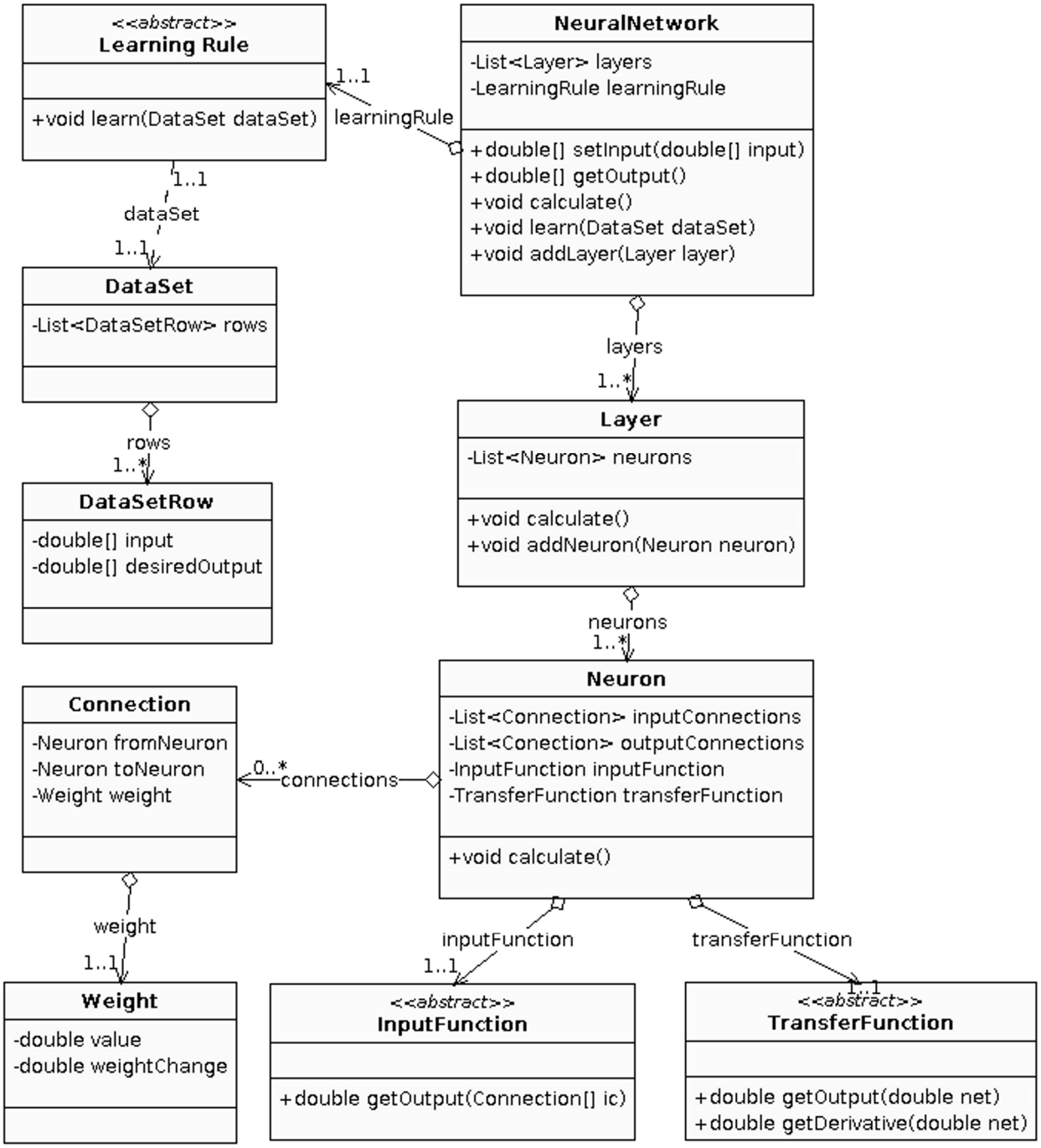


Fig. 1 *The core class diagram of the Neuroph framework*

In general, the Neuroph framework provides a set of base classes (Fig. 1), which can be easily extended by using inheritance and overriding specific methods. This approach provides the same general structure for different types of neural networks. The main classes, methods, and extension points are briefly explained in the rest of this section, while more technical details are available in section 4.

1) The **NeuralNetwork** class is the base class for all types of neural networks. It contains a collection of neuron layers and a learning rule. It provides methods that represent a common public interface for all types of neural networks, which are:

setInput() - sets the network input

calculate() - calculates the network output. The default implementation is sequential calculation of the network layers. For different calculation strategies, this method should be overridden and it can delegate calculation to other classes.

getOutput() - returns the network output

learn(DataSet data set) - runs a learning algorithm with the specified data set . This method delegates the learning algorithm to some implementation of the *LearningRule* class.

Specific types of neural networks are defined by extending this class, implementing the method that creates a specific network architecture (layers, neurons and connections), and setting the appropriate learning algorithm based on the *LearningRule* class.

2) The **LearningRule** class is an abstract base class for all types of learning algorithms. The extension point is the abstract method *learn()*,which should be implemented by subclasses in order to create specific learning algorithms. It also provides methods to control learning (stop, pause, resume), a learning event mechanism, and methods that can be overridden in some learning stages (onStart, onStop).
This class is inherited by *UnsupervisedLearning* and *SupervisedLearning*, which are further inherited by specific learning rules like *LMS*, *Backpropagation*, etc.

3) The **DataSet** class represents a collection of data that the network should learn during the training process. It can contain both supervised and unsupervised training data.

4) The **Layer** class is a collection of neurons, and also a base class for different types of layers. In most of cases, the basic layer class is enough. It provides methods to add to and remove neurons from a layer, and the *calculate()* method that calculates the output of all neurons in a layer. The default implementation does sequential calculation of neurons outputs, and different strategies for calculation can be implemented by sub classing and overriding the *calculate()* method.

5) The **Neuron** class is a base class for all types of neurons. It has an input function and a transfer function, collections of input and output connections to other neurons and the *calculate()* method that delegates calculation of the neuron output to its input and transfer functions. Different calculation behavior can be implemented by overriding the *calculate()* method.

6) The **InputFunction** class is an abstract class used as a base class for all types of a neuron's input functions. It has one abstract method - *getOutput(Connection[] inputConnections)* - which has an array of input connections as an input parameter, and returns the result/output of the input function.Specific input functions are implemented by extending this class and implementing the method *getOutput().* The framework provides implementations of many commonly used input functions: weighted sum, distance, and, or, min, max, etc.

7) The **TransferFunction** class is an abstract class used as a base class for all types of a neuron's transfer functions. It has two important methods: the abstract method *getOutput(double net)*,which should return the output of a specific transfer function, and *getDerivative(double net)*, which should return the first derivative of a specific function. The framework provides implementations of many commonly used transfer functions, like: step, sigmoid, tanh, gaussian, linear, ramp, log, sin, etc.

8) The **Connection** class represents a weighted connection between two neurons. It holds references to

from and *to* neurons, and the connection weight, which is an instance of the *Weight* class, so the single *Weight* instance can be shared by multiple connections (which is an important feature of architectures like Convolutional networks).

9) The **Weight** class stores a value of the connection weight (as a real number) and the weight change (which is calculated by the learning rule).It also may contain additional data used by the learning rule and related to that weight (so called *trainingData*). This class does not have subclasses, and typical scenarios do not require this class to be extended.

10) The **PluginBase** class is a base class of the Neuroph plugin system, with the purpose to provide an extensible mechanism to create application-specific functionalities and keep them separated from the neural-network-related classes. In practice, this is done by creating an application-specific plugin class that inherits *PluginBase* and adds application-specific methods (like, for example, *ImageRecognitionPlugin*). This class is used to separate application-specific logic from neural-network-related logic, which is identified as one of the main problems with neural network development and reuse.

As of version 2.9, Neuroph supports the following types of neural network architectures: Adaline, Perceptron, Multi Layer Perceptron [17], Hopfield [18], Bidirectional Associative Memory [19], Kohonen [20], Instar [21], Outstar [21], Competitive Network [22], MaxNet [22], Radial Basis Function Network [23], Neuro Fuzzy Perceptron [24], Hebbian Network [25], and Convolutional Network [26]. From this list of supported neural networks, it can be seen that the basic Neuroph design supports creation of the following types of neural network architectures: feed forward, recurrent, fully connected, competitive. Thanks to flexibility and reusability of the base components, it also supports creation of modified custom components and connectivity patterns between neurons.

Neuroph supports the basic supervised and unsupervised learning rules and their variations, which are used by the networks listed above. The supported learning rules include: LMS, Perceptron Learning, Delta Rule, Backpropagation [17] , Resilient Propagation [27], Dynamic Backpropagation, Hebbian Learning [25], Anti Hebbian Learning [28], Oja Learning [29], Instar Learning [21], and Outstar Learning [21].
From the list of supported types of learning rules, it can be seen that the base Neuroph design supports creation of the following types of learning rules; Unsupervised (Hebbian, Kohonen, Competitive), Supervised (LMS, Backpropagation, etc.) and single-pass like Hopfield Learning [18] .

The working implementation of a wide variety of neural network architectures and learning rules supported by Neuroph are all based on the same base components, which demonstrates the flexibility and reusability of the design.

2.2. Neuroph Studio

Neuroph Studio is an integrated development environment for neural networks, with a graphical user interface (GUI); it is built on top of the NetBeans Platform. It provides tools for creating and analyzing neural networks with the Neuroph framework. The main window of the Neuroph Studio application is shown in Figure 2.

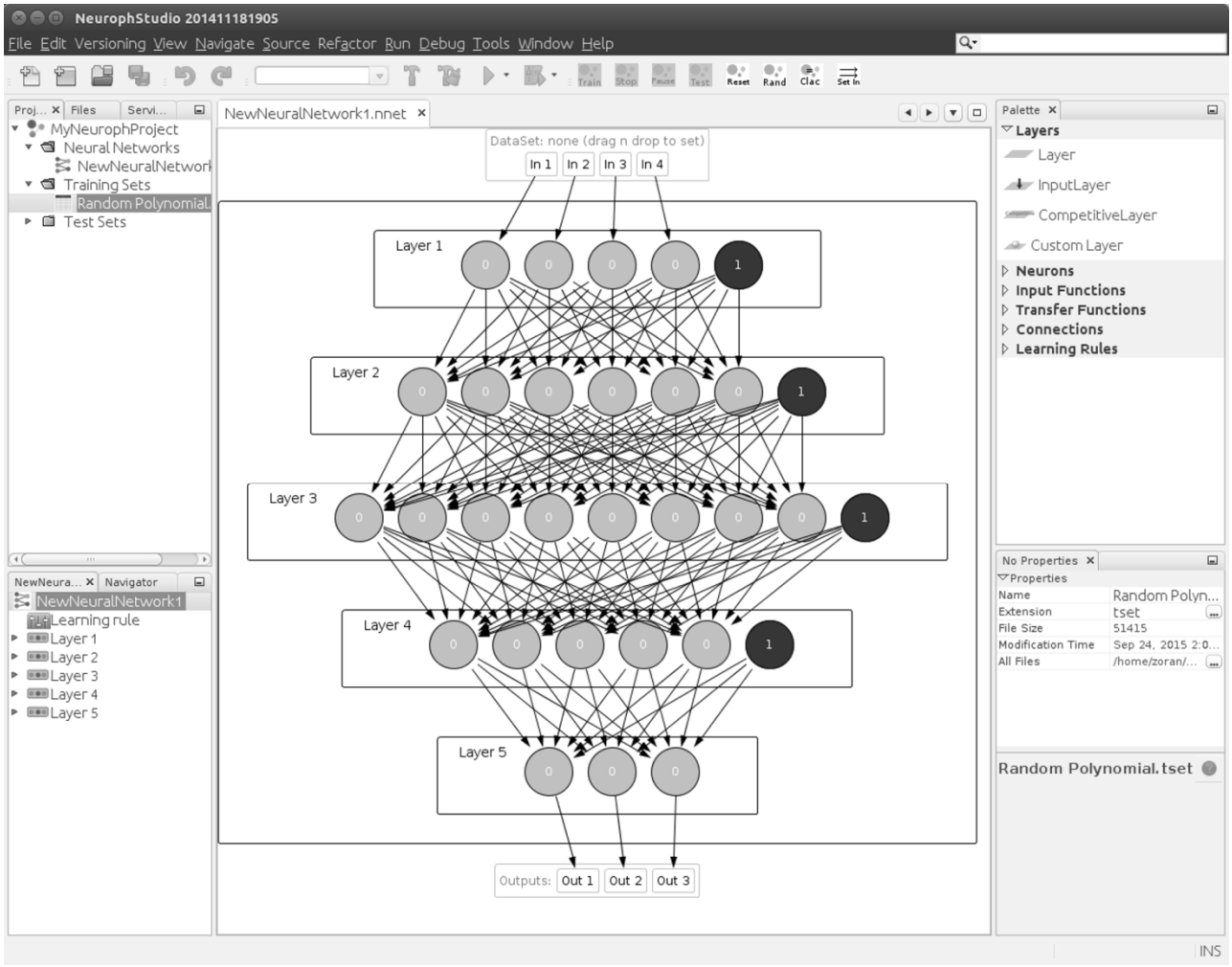


Fig. 2 *The main window of Neuroph Studio development environment*

Neuroph Studio provides a wizard-based workflow for neural network development, which is familiar to software developers, and guides the user through a series of steps/dialogs in order to accomplish a specific task. It also provides visual tools to create and analyze neural networks and data sets.
Neuroph Studio provides the following main components:

1. Project System – manages a set of neural network and data set files for some specific problem
2. Wizard System – provides a set of wizards for creating neural networks and data sets
3. Visual Editor & Palette – visual tool for editing neural network components and connections using drag 'n' drop from component palette
4. Explorer View- provides a tree-like view of neural network components
5. Properties View – provides all settings of the currently selected neural network component
6. Visualization System – provides various 2D and 3D visualizations of different neural networks and data set s

7. Training System - a set of dialogs for setting all training parameters and monitoring training
8. Cross-validation tool – a tool for automated cross-validation
9. Application-specific tools – various tools for application-specific domains like image recognition, optical character recognition, EEG signal recognition etc.

A detailed description of these Neuroph Studio components is given in application examples presented in section 3.

NetBeans Platform, which is used as a base for Neuroph Studio, is a generic framework for building rich, complex Java desktop applications. It provides flexible modular architecture, rich set of graphical user interface components, and many other important features that can be easily reused. NetBeans Platform is an open-source software, developed by Oracle Corporation. Neuroph Studio adds neural-network-specific tools based on the Neuroph framework to NetBeans Platform in the form of additional NetBeans modules/plug-ins.
This also makes easy for Neuroph to integrate with other software based on NetBeans Platform, for example NetBeans Java IDE. This way, Neuroph Studio provides an integrated neural network and Java development environment, and make it easier for software developers to create, use, and integrate neural networks in their software projects.

Hardware and software requirements

Neuroph Studio runs on operating systems that support Java Virtual Machine (Windows, Linux, Mac), and meet the following recommended requirements:

CPU: 2.6 GHz Intel Pentium IV or equivalent,
Memory: 2Gb
Java version: 1.8

Note that default settings for assigned memory for Java Virtual Machine in Neuroph Studio are pretty low (only 64Mb), but if Neuroph Studio becomes unstable, or 'Out of memory error' occurs, more memory should be assigned to it using -J-Xmx switch in configuration file *[NeurophStudioHomeDir]/etc/neurophstudio.conf*[8]

3. Application Cases

This section shows how Neuroph software is used to solve some typical machine learning problems, like classification and image recognition. Three application cases show how to use visual tools from Neuroph Studio to create and train neural networks, and then the same problems are solved in Java code. The first demo is educational and its focus is on the training procedure. The classification problem data set and the neural network are generated automatically, with the smallest number of settings, and the user can easily train the network using the training dialog. This demo also provides visualization tools which enables better understanding of neural network training and operation.
The second demo is more realistic and shows how to import a data set from an external file, and create and customize neural network using a specialized wizard. This is demonstrated using the Iris classification problem.

[8] https://sourceforge.net/p/neuroph/discussion/862858/thread/17bf78d7/

The third demo shows how to use Neuroph in a specific application domain, which is in this case image recognition.

3.1. Basic Classification Demo

This demo shows how to solve simple classification problems using the Neuroph Studio GUI. Neuroph Studio provides wizard-based tools to generate sample data set s and neural networks, for simple classification problems, and then train and test the neural networks. NeurophStudio also provides visualization tools that help the user to get a better understanding of the problem, the network architecture, the learning algorithms and the network operation.
This demo is an educational example, useful for teaching neural networks, explaining basic concepts, workflow, visualization of the problem (data set) and Multi Layer Perceptron operation (learning and classification).
The problem is defined as follows: Create and train Multi Layer Perceptron neural network to classify a set of two-dimensional points (vectors). The set of two-dimensional points is generated automatically, as a sample data set , where each point belongs to one of the two possible classes (red or blue). The class of a point is determined during the data set generation, using simple shape patterns: if the point belongs to the inner part of the shape then it is blue, otherwise it is red. There are several different shapes that can be used: ellipse, circle, square, diamond, moon, ring, etc.

Brief overview of the procedure

The problem described above can be solved with Neuroph Studio in the following steps:
1. Creating a Multi Layer Perceptron Classification Sample Project
2. Generating a sample data set for classification, based on a predefined shape.
3. Creating a Multi Layer Perceptron neural network, with a selected predefined architecture.
4. Training the neural network. Apply the training procedure to the neural network using the generated data set .

These steps can be easily executed and visualized using Multi Layer Perceptron Classification Sample in Neuroph Studio. The text that follows explains the main points with screenshots and step-by-step instructions for running this sample.

Step 1. Creating a Multi Layer Perceptron Classification Sample Project

Neuroph Studio has a project-based workflow, which means that the first thing that needs to be done when working on some problem is to create a project. A project is a logical set of neural networks, data set s, and specific tools available in specific types of projects. Different types of projects are created by using the 'New Project' wizard and selecting the project type.
The Multi Layer Perceptron Classification Sample project wizard (Fig. 3) is launched from the main menu:
Main Menu > File > New Project > Samples > Neuroph > Multi Layer Perceptron Classification Sample

After the 'next' button is clicked, the wizard asks for the name and location of the project, and the project will be created in a folder at a specified disk location. There are three main components that will be available upon the project creation (Fig. 4):

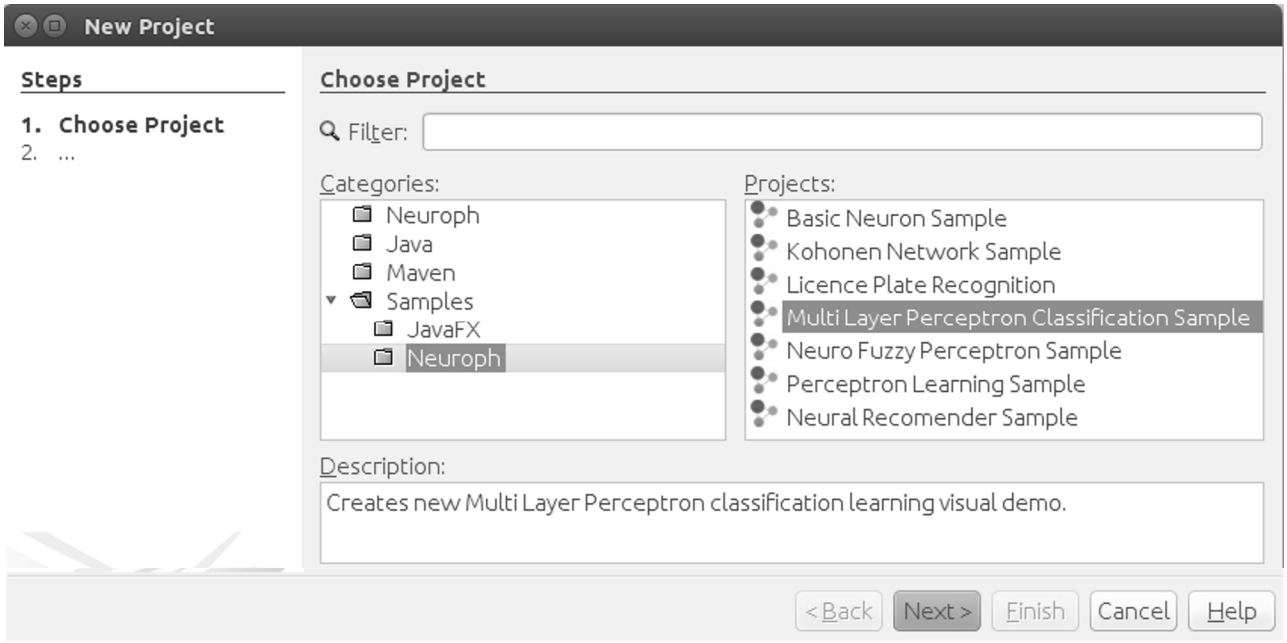


Fig. 3*Multi Layer Perceptron Classification Sample Project Wizard*

1) Project Window - contains neural networks and data set s organized into folders
2) Visualization Window –provides 2D visualization of the problem data set and animated neural network learning
3) Sample Controls Window –provides controls for creating data set , neural network and setting visualization options.

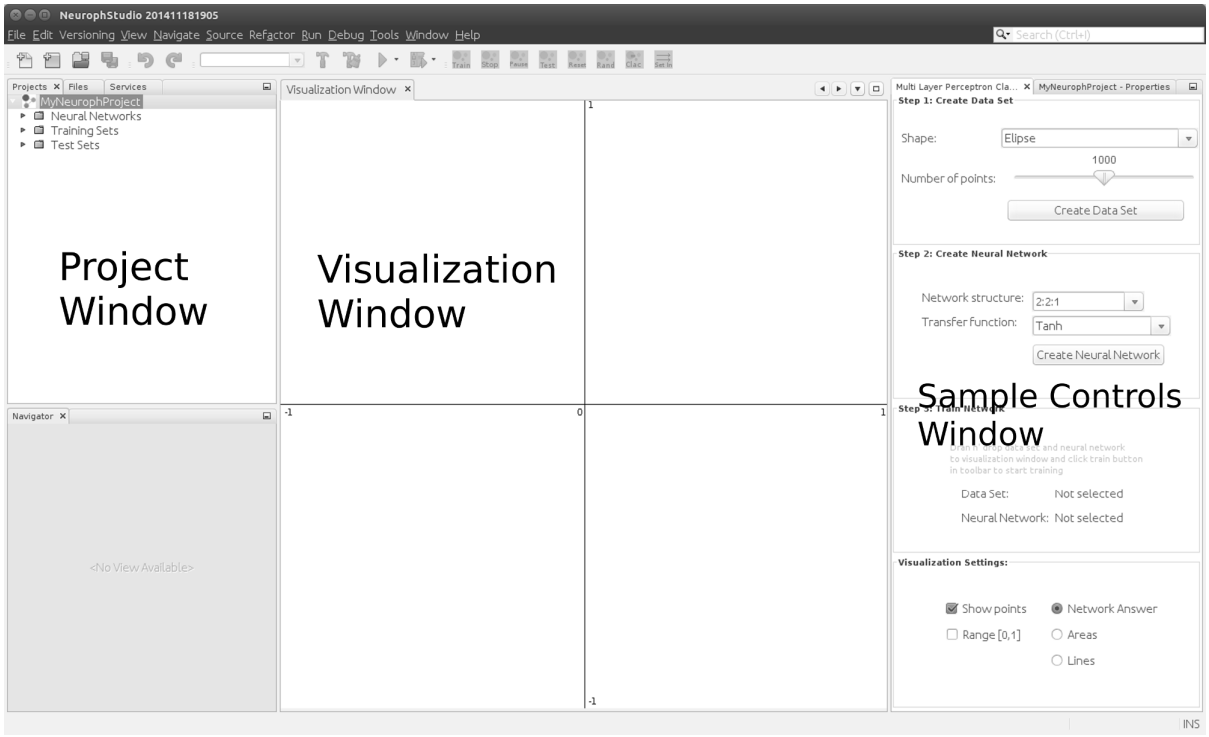


Fig. 4 *The main view of Multi Layer Perceptron Classification Sample*

Step 2. Creating a data set

To create a data set, select one of the Shape options (for example Diamond) in Sample Controls Window and click the 'Create Data Set' button. It is also possible to set the number of points that will be created (Fig. 5).

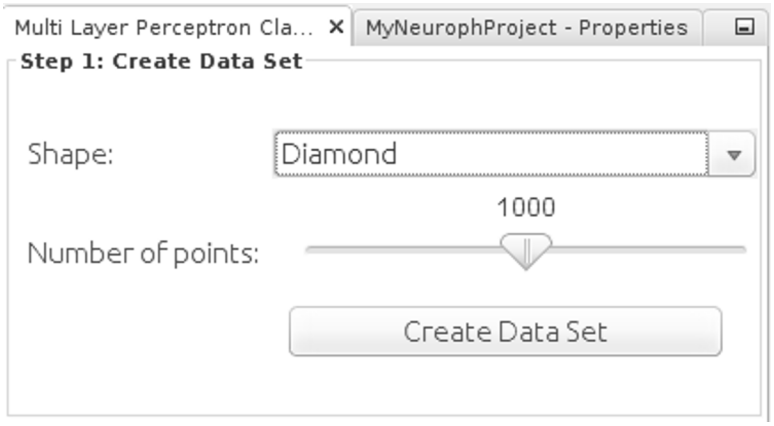


Fig. 5 *Create data set controls*

A data set for the selected shape will be created under the Training Sets folder in the Project Window, and visualized in the visualization window (Fig. 6).

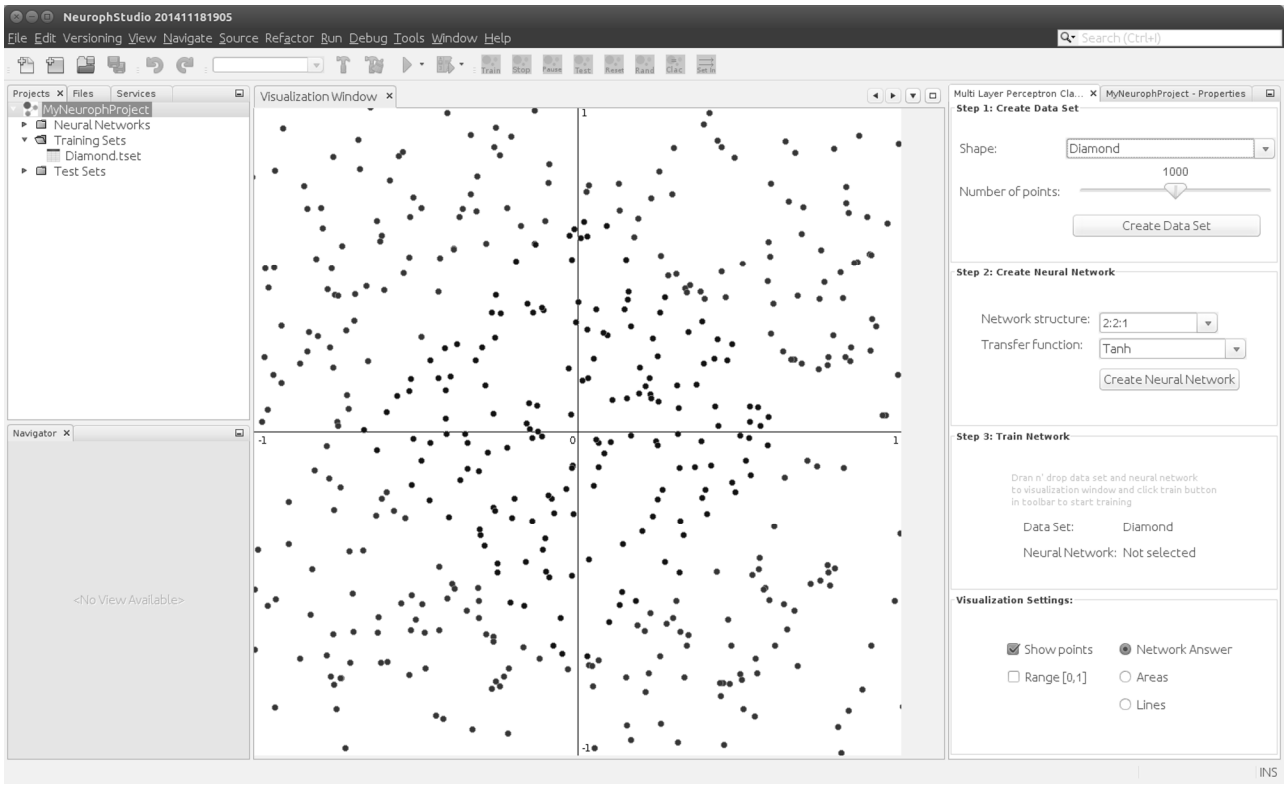


Fig. 6 Visualization of c*reated diamond data set*

Each 2D point in the generated data set has two values in the interval [-1, 1] and the classification class that can be either 1 (blue, or inner shape point) or 0 (red or outer shape point). The generated data set can also be viewed as a table with numerical values, by clicking the data set in the project window (Fig. 7).

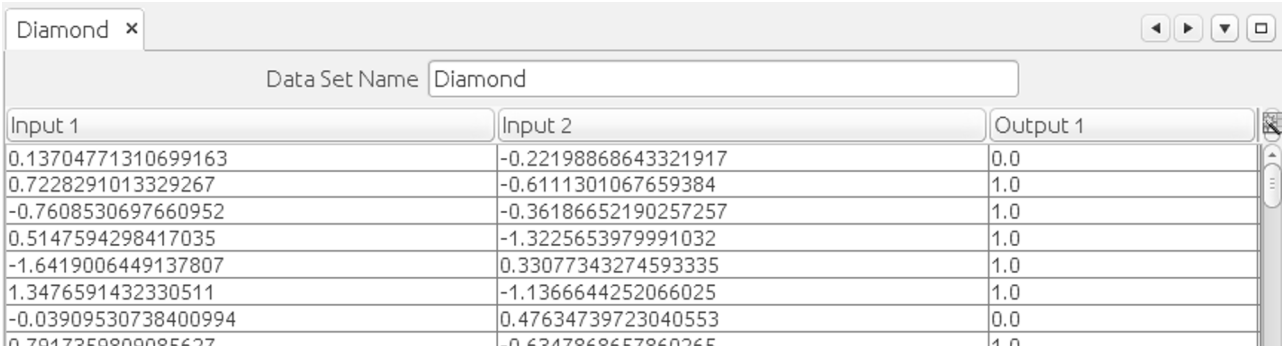


Fig. 7 *Data set table view*

Step 3. Creating a Multi Layer Perceptron neural network

To create a Multi Layer Perceptron, select the network structure and transfer function in the Sample Controls Window and click the 'Create Neural Network' button (Fig. 8).

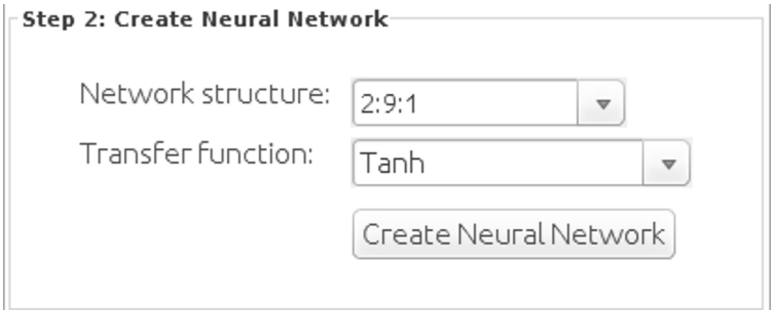


Fig. 8 *Controls for creating a neural network*

The network structure corresponds to the number of neurons and layers in a neural network. For example, 2:9:1 means that the neural network has two neurons in the input layer, nine neurons in the hidden layer, and one neuron in the output layer. In this example, all proposed architectures have two neurons in the input layer, and one neuron in the output layer, since the input is two-dimensional and a single binary [0, 1] classification as the output.

A Multi Layer Perceptron neural network will be created under the 'Neural Networks' folder in the Project Window, and it can be opened using the Visual Editor tool in the central window (Fig. 9). Visual Editor provides a visual, component-based view of a neural network architecture: layers, neurons and connections. It enables inspection and modification of the individual components.

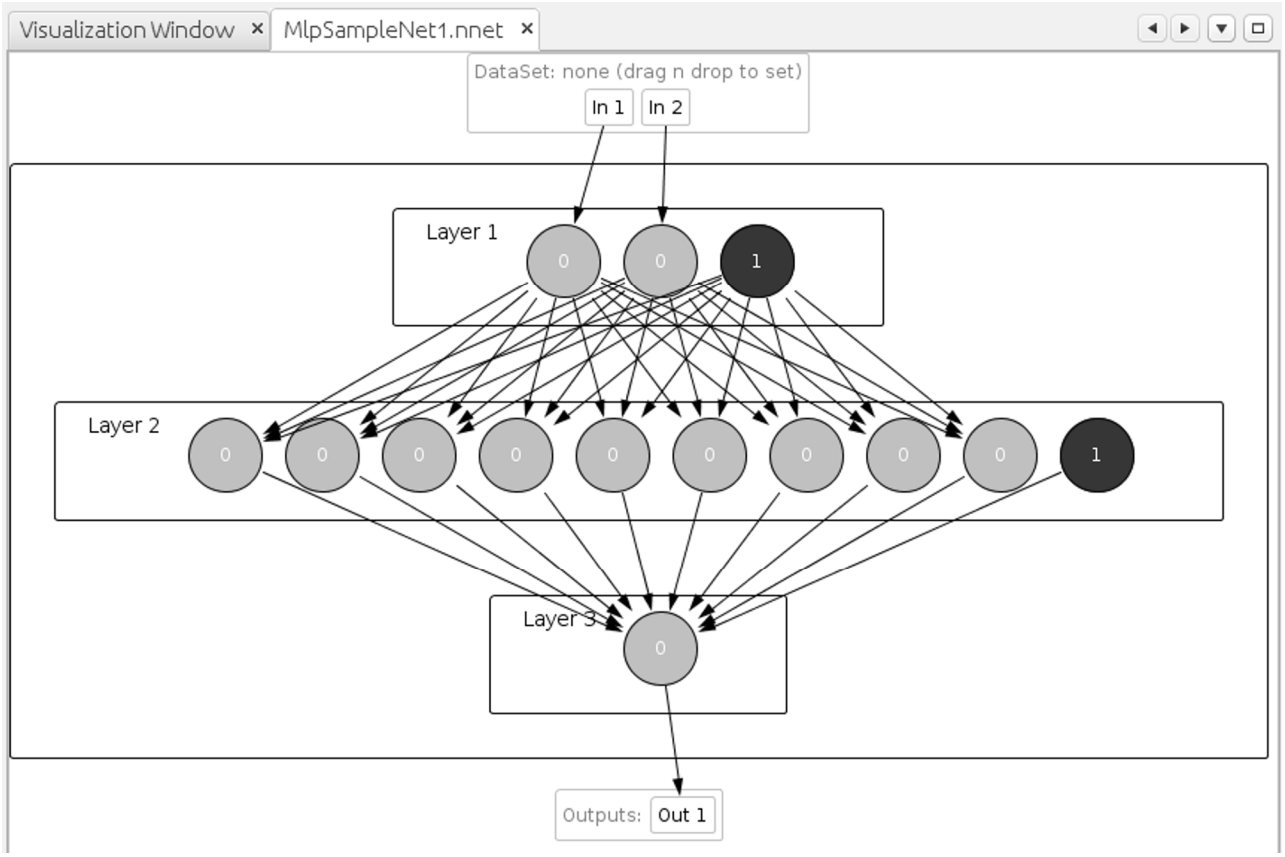


Fig. 9 *Visual neural network editor*

Note that individual additional neurons in the input and hidden layers shown in Fig. 9 represent bias neurons that are added by default. Bias neurons represent constant input for all neurons in next layer, that helps faster convergence of Backpropagation learning rule.

Step 4. Training a neural network

To start the neural network training with some data set and observe the training process, the neural network has to be dragged 'n' dropped from the Project Window to the data set visualization window in the center, to enable the train button in the toolbar. By dropping the neural network in the visualization window, the user indicates that he wants that neural network to learn the visualized data set .

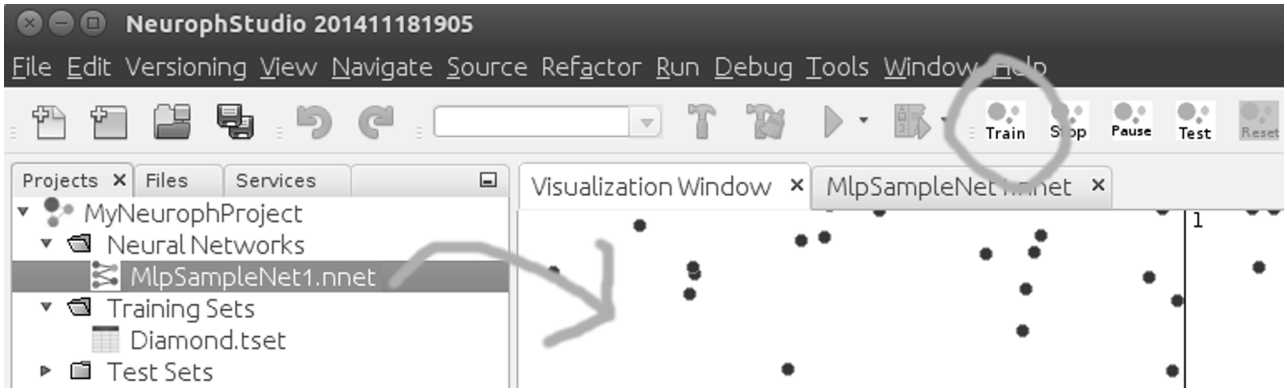


Fig. 10 *Drag 'n' drop the neural network and start training*

Clicking the 'Train' button in the toolbar (Fig. 10) opens the training dialog (Fig. 11).

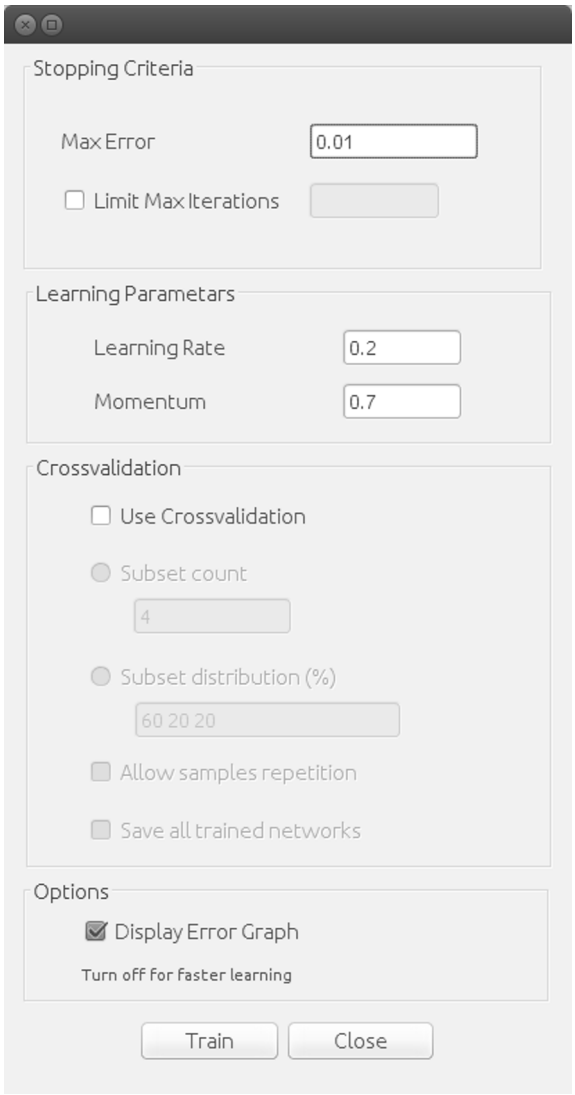


Fig. 11 *Training dialog with default settings*

The training dialog provides various settings for the learning algorithm like Max Error, Learning Rate and Momentum. For the purpose of this demo, the provided default values are accepted. Clicking the 'Train' button starts the training procedure: it opens real-time total network error graph (Fig. 12) and displays the network operation in the visualization window.
Note that the dialog also provides some advanced settings for the cross-validation procedure, which is described in more detail in Section 3.2.

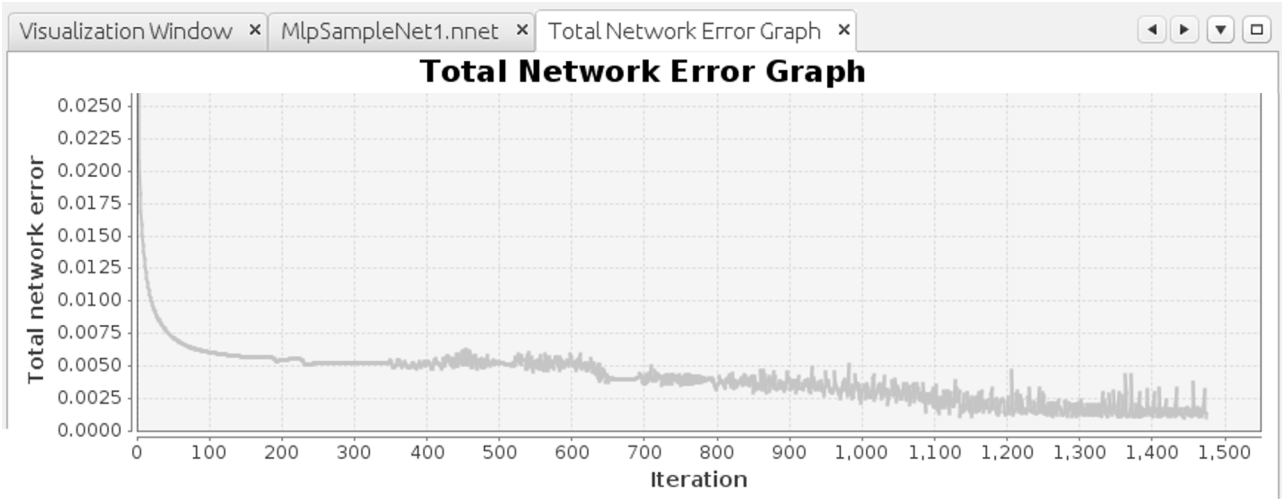


Fig. 12 *Total network error graph during training*

Total network error graph shows the value of total network error during the training iterations. During the training, this is displayed on real-time scrolling graph, and at the end of the training it is displayed for the entire training. The fundamental principle of all supervised learning algorithms is minimization of a total error, and this graph helps to get a better understanding of that process.
The network operation during the training can be observed in real time in the visualization window. This way, it is possible to observe how classification boundaries are changing during the training. The final training result shown in Fig. 13 shows how decision boundaries have formed a diamond shape, which was the one used for training in this example. This picture shows how the trained network 'sees' the provided data set .

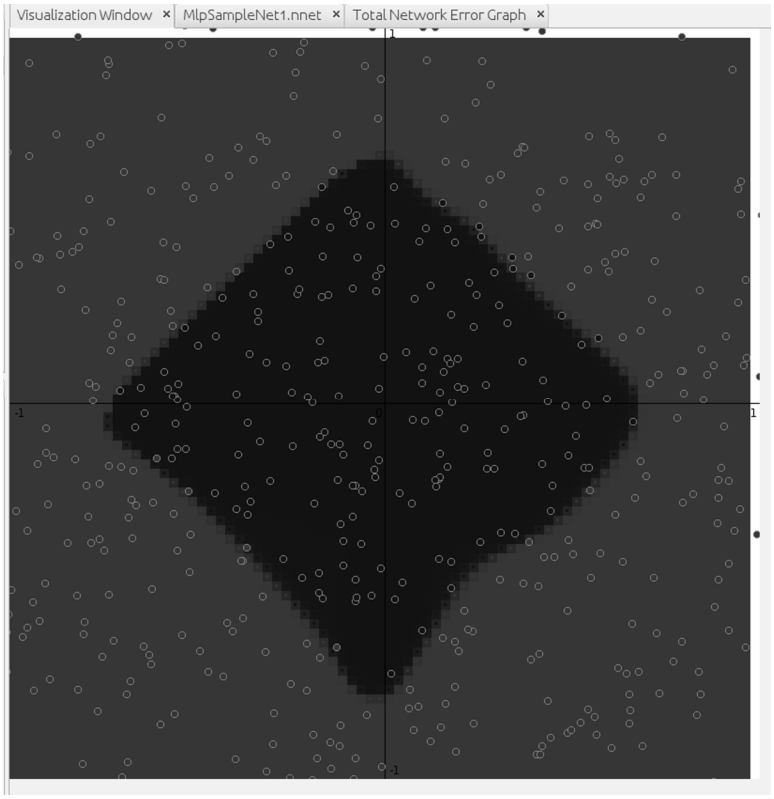


Fig. 13 *Visualization of the data set and classification by the trained network*

Figure 14 shows how the hidden neurons are creating decision areas, and Figure 15 shows how decision lines of the hidden neurons are dividing the input space.

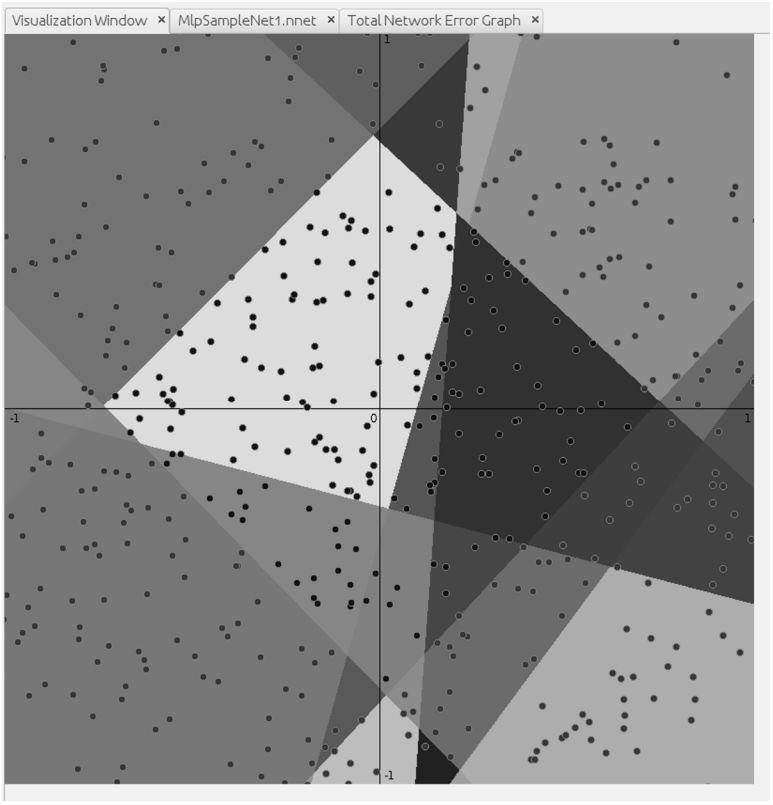


Fig. 14 *Visualization of areas created by neural network classifier*

By using these visual tools, the user can easily experiment with different neural network architectures and learning rule settings, in order to see how they affect training and classification boundaries.

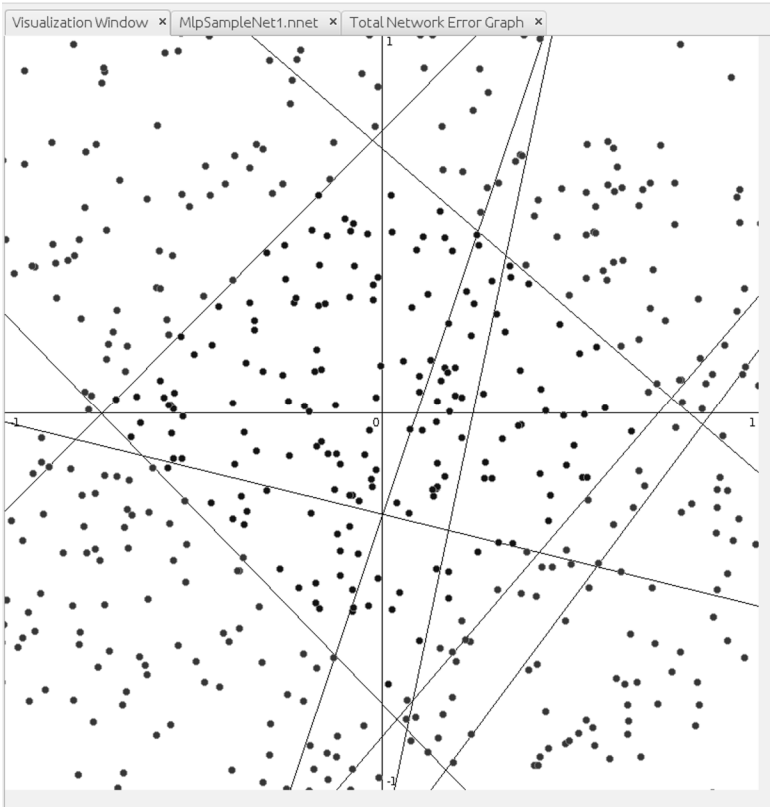


Fig. 15 *Visualization of the decision lines created by the neurons in the hidden layer*

This classification demo shows the basic components and workflow when working with neural networks in Neuroph Studio. It also shows how visual tools can help in getting a better understanding of the basic neural network components and operation. This tool is used for teaching neural networks in the course on Intelligent Systems at the University of Belgrade. It clearly demonstrates high-level concepts, and using visual representation helps students to understand the main principles behind neural network classifiers.

3.2. Iris Classification Example

This example shows how to use Neuroph to solve a classification problem, starting from an external data set, using Neuroph Studio GUI, and also in Java code. The example shows how to import a data set , create and train the corresponding neural network, and apply a performance evaluation procedure. This example uses the Iris flowers data set for demonstration purposes.

Problem description: Iris flower classification problem
Fisher's Iris data set (Fisher, 1936) is a well-known data set in the pattern recognition literature. The data set contains 3 classes with 50 instances each, where each class refers to a type of the iris plant. One class is linearly separable from the other two; the latter are not linearly separable from each other. This data set has become a typical test case for many classification techniques in machine learning.
The data set contains the following attributes:
1). sepal length in cm
2). sepal width in cm

3). petal length in cm
4). petal width in cm
5). class:
- Iris Setosa
- Iris Versicolour
- Iris Virginica

In order to solve this problem using Neuroph, a Multi Layer Perceptron neural network, which is commonly used as a classifier, needs to be trained with the provided Iris flowers data set .

This problem can be solved using the Neuroph Studio GUI, through the following steps:

1. Create a new empty Neuroph project

2. Import data set . Create a data set object from the given file in CSV format;

3. Create a Multi Layer Perceptron neural network.

4. Train the neural network. Apply the training and testing procedure including cross-validation procedure using imported data set .

3.2.1. Iris Classification Example Using the Neuroph Studio GUI

Step 1. Create a new empty Neuroph project:

When working with neural networks and data set s, a Neuroph project system provides a folder structure and context. A wizard for creating a new, empty Neuroph project (Fig. 16), is launched from the main menu: *Main Menu* > *File* > *New Project* > *Neuroph* > *Neuroph Project*.

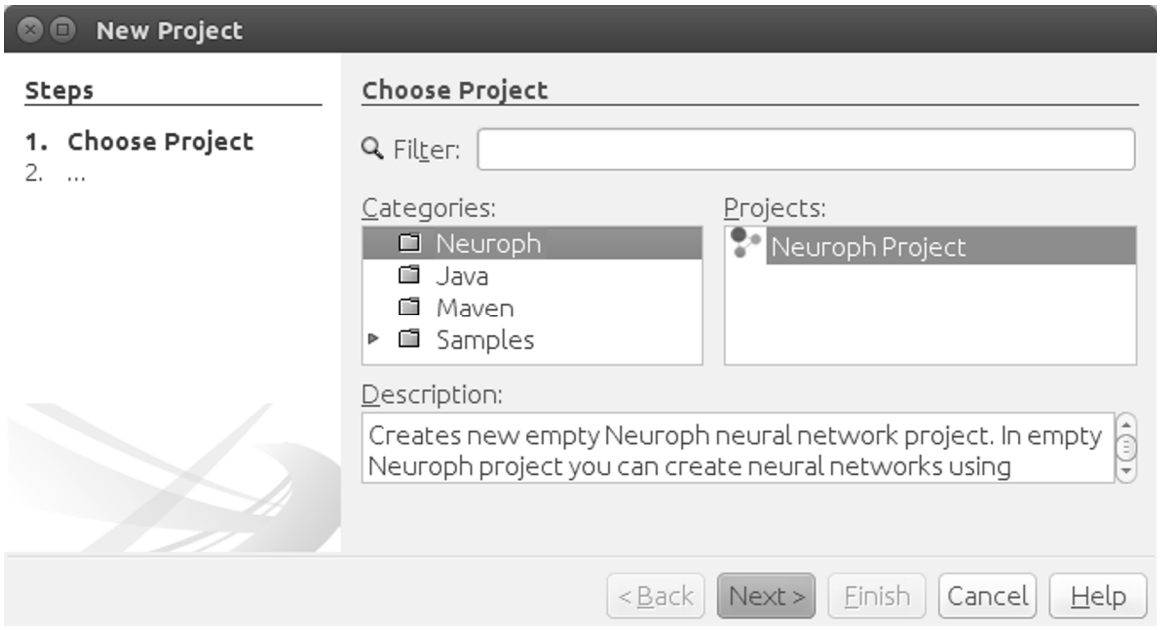


Fig. 16 *New Neuroph Project Wizard*

In the next step, the wizard asks for the name and location of the project, and then creates the project in

a folder with the project name, at a specified disk location (Fig. 17). In the empty Neuroph project, the user can import data set s to create training sets, create neural networks using the wizard, and train the neural network.

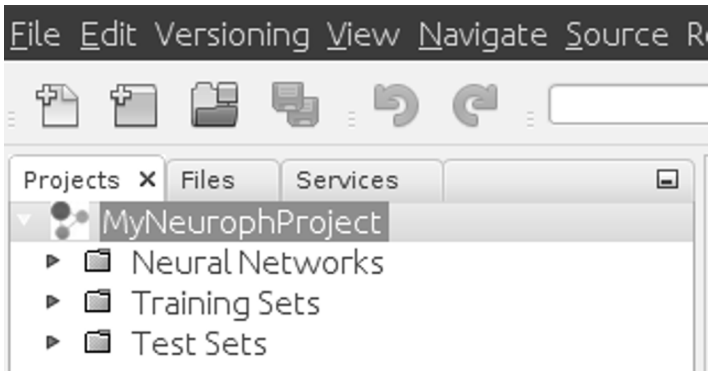


Fig. 17 *Neuroph project structure*

Step 2. Create classification data set by importing it from a CSV file

The original Iris Classification data set available from the UCI Machine Learning repository (*https://archive.ics.uci.edu/ml/data set s/Iris*) has four numerical and one nominal attribute, which corresponds to the instance class (as described in the problem description section). In order to import and use this data set in Neuroph Studio, the original data set must be preprocessed, so the numerical features are normalized to the range [0, 1] by using the max normalization, and the nominal class attribute is converted into three binary numeric attributes (since there are three nominal classes). This preprocessing is required, since neural networks operate with values in the range [0,1]. Table I shows the original and normalized values for a single data set row.

	Inputs				**Output(s)**		
Original values	5.1	3.5	1.4	0.2	Iris-setosa		
Normalized values	0.6455	0.7954	0.2028	0.08	1	0	0

Tab. I *Original and preprocessed data set values*

Note that data set s can also be normalized in Neuroph Studio, but in order to keep the focus of the tutorial, the prepared data set is provided in supplementary material[9] in file *Iris-data set - normalized.txt*.

The preprocessed file can be imported in the Neuroph project using the New Data set wizard, which is launched from the main menu: *Main menu > File > New > Neuroph >Data set* (Fig. 18).

[9] http://ai.fon.bg.ac.rs/wp-content/uploads/2015/12/SupplementaryMaterial.zip

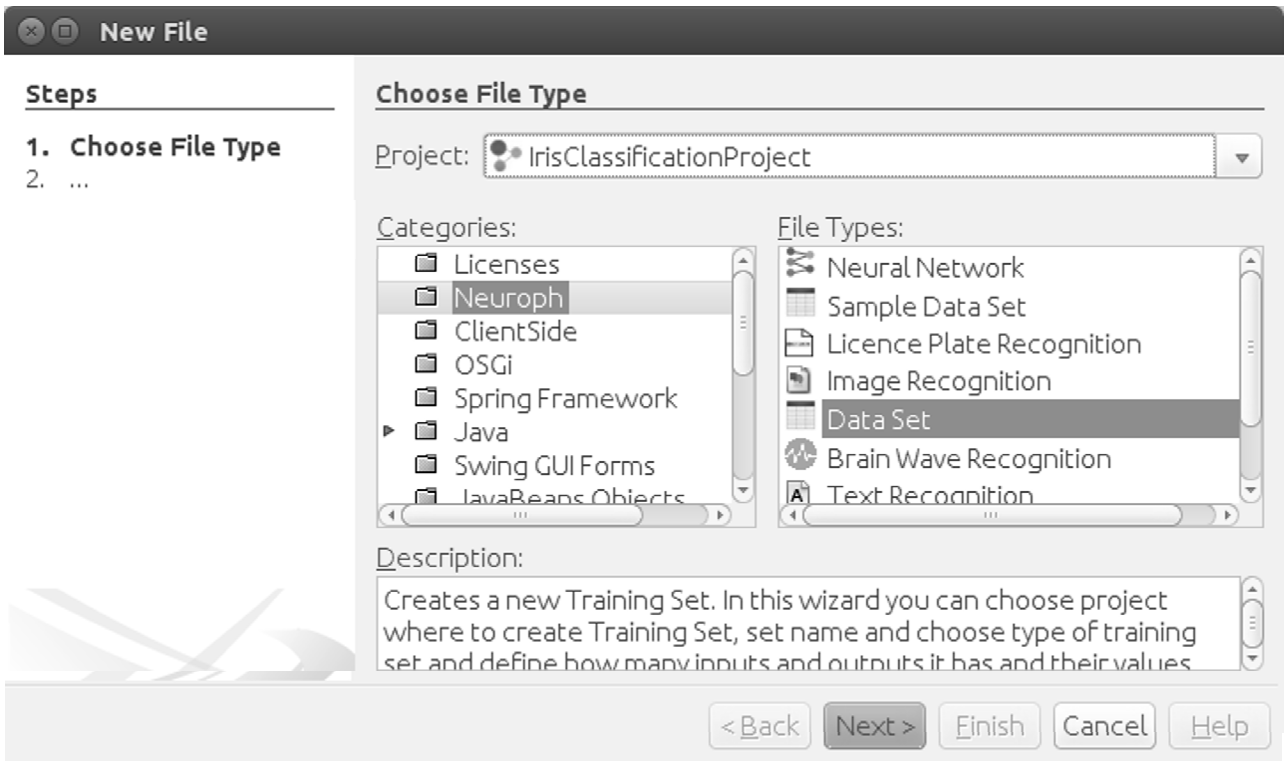


Fig. 18 The *New Data set wizard*

In the next step (Fig. 19), the wizard asks for the information required for creating the data set in Neuroph Studio from the given CSV file.

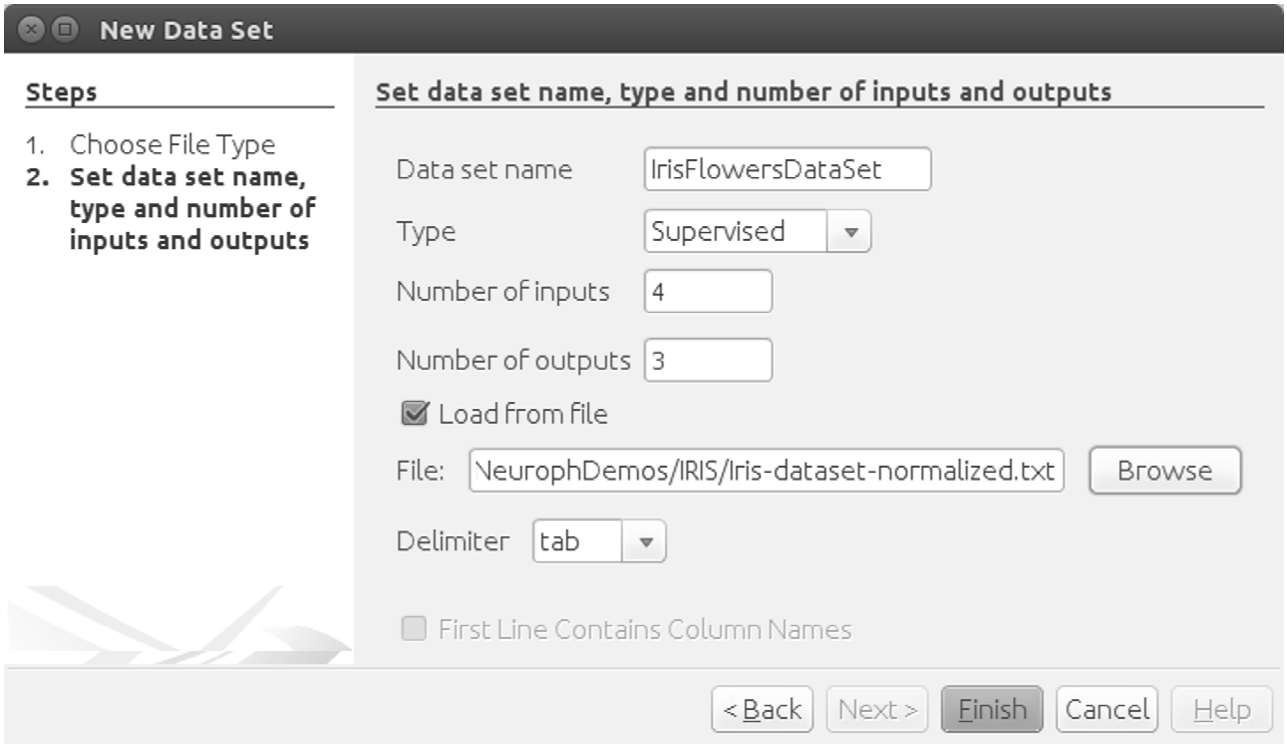


Fig. 19 The *New Data set wizard parameters*

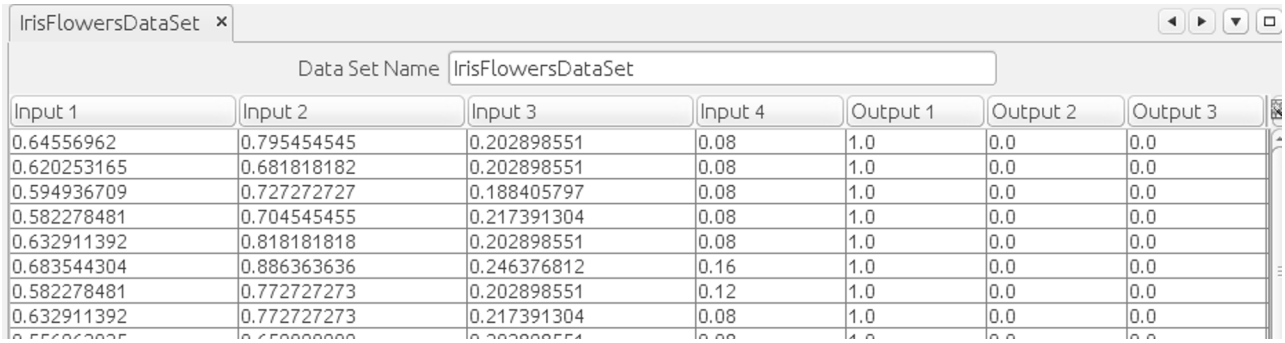
The parameters that need to be specified when importing the data set are described in the Table II.

Data set name	Name of a data set
Data set type	Data set type; can be supervised or unsupervised. For Iris classification, this value is *Supervised*, since there is a target class specified for each row in the data set .
Number of inputs	Number of input values in each data set row. For Iris classification, this value is 4, since there are 4 attributes in the data set .
Number of outputs	Number of output values in each data set row. Outputs are specified only for supervised data set s, and they correspond to neural network outputs and target values. For Iris classification, this value is 3, since the nominal class attribute is binarized to 3 outputs in the preprocessed data set.
File	The actual data set file on disk (in CSV format).
Delimiter	Value delimiter in the CSV file (possible values are coma, space, tab or semicolon)

Tab II The *Data set wizard parameters*

Figure 19 shows the settings for importing the Iris Classification data set from the preprocessed CSV file provided in the supplementary material.

The imported data set is shown in the project window, and it can be opened in tabular form (Fig 20) in order to see the imported values, or visualized in 2D or 3D using some of the Neuroph Studio visualization tools.



Input 1	Input 2	Input 3	Input 4	Output 1	Output 2	Output 3
0.64556962	0.795454545	0.202898551	0.08	1.0	0.0	0.0
0.620253165	0.681818182	0.202898551	0.08	1.0	0.0	0.0
0.594936709	0.727272727	0.188405797	0.08	1.0	0.0	0.0
0.582278481	0.704545455	0.217391304	0.08	1.0	0.0	0.0
0.632911392	0.818181818	0.202898551	0.08	1.0	0.0	0.0
0.683544304	0.886363636	0.246376812	0.16	1.0	0.0	0.0
0.582278481	0.772727273	0.202898551	0.12	1.0	0.0	0.0
0.632911392	0.772727273	0.217391304	0.08	1.0	0.0	0.0

Fig. 20 *The imported data set*

Data set visualization is important for users in order to get a better understanding of the problem. Figure 21 shows the Iris classification data set visualized with 2D scatter graph, where classification classes can be observed with respect to the chosen input attributes.

Neuroph Studio supports the following types of data set visualization(which can be launched by right-clicking the data set in the project view):

- 2D Scatter, 2D Line
- 3D Scatter, 3D Surface, 3D Histogram

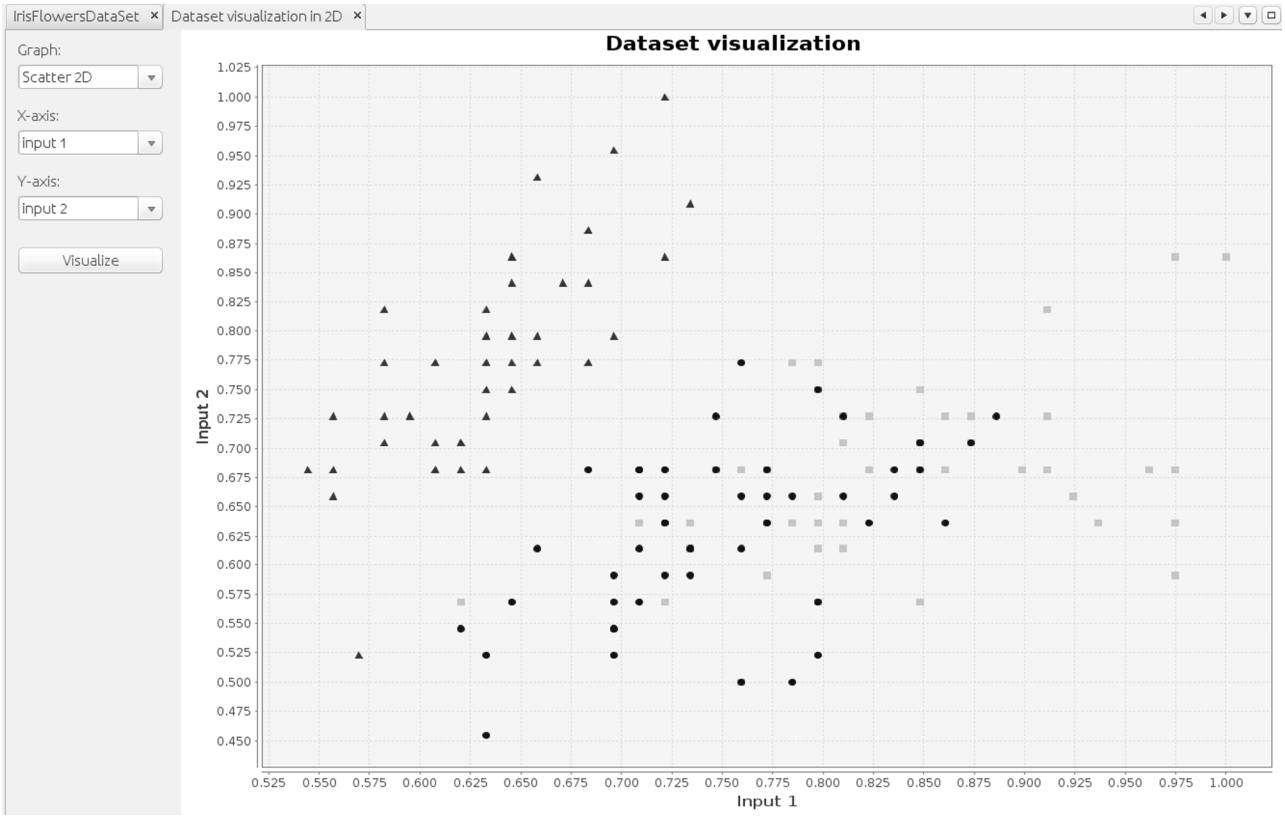


Fig. 21 *Scatter 2D visualization of the Iris Classification data set*

Step 3. Creating a new Multi Layer Perceptron neural network using a wizard

A new Multi Layer Perceptron neural network that will be trained with the Iris classification data set can be created with the *'New Neural Network'* wizard, which is launched from the main menu (Fig 22):

Main Menu > File > New > Neuroph > Neural Network

Figures 23-24 show screen-shots of the New Neural Network wizard steps for creating the Multi Layer Perceptron network type.

In the first step (Fig. 23), the wizard expects a name and neural network type to be provided. In this example, the neural network type is Multi Layer Perceptron.

In second step (Fig. 24), the wizard expects the parameters specific to Multi Later Perceptron neural network type to be entered.

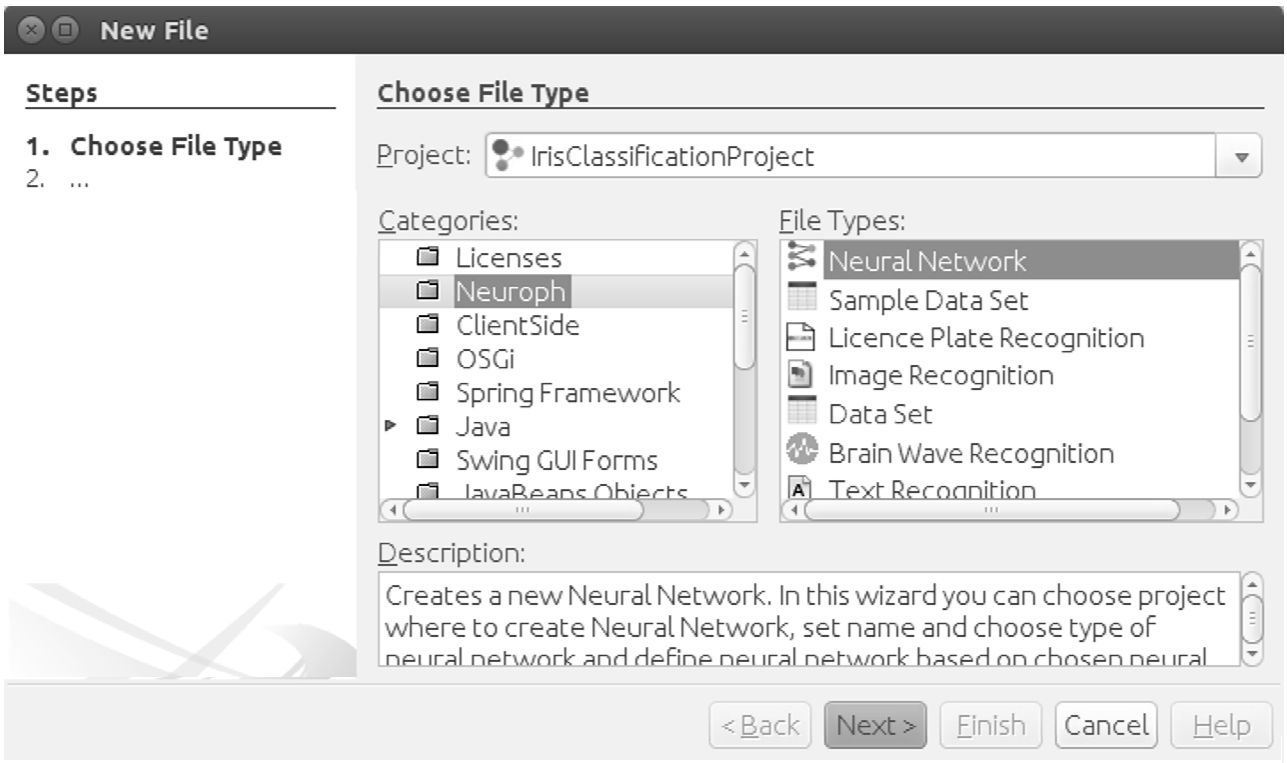


Fig. 22 *Starting the New Neural Network Wizard*

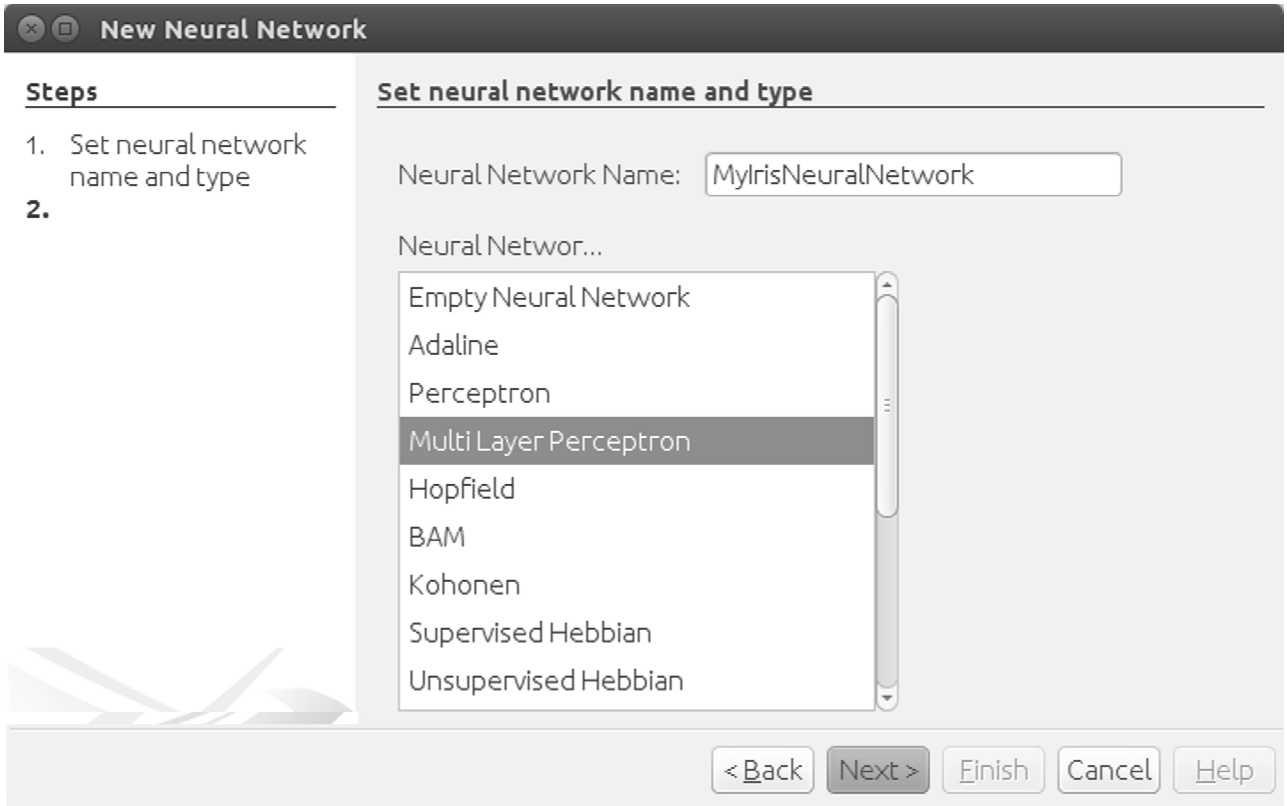


Fig. 23 The *New Neural Network Wizard step 1*

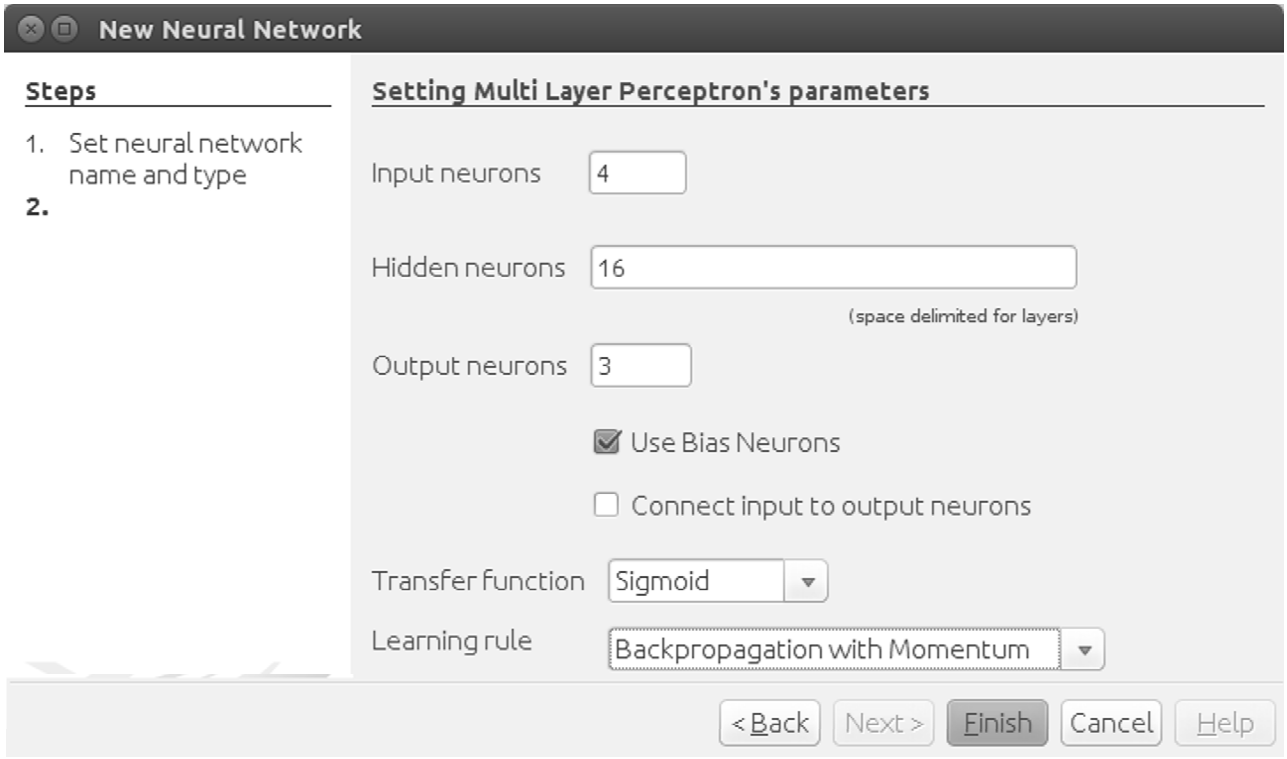


Fig. 24 *The New Neural Network Wizard step 2*

Parameter	**Description**
Input neurons	Number of network inputs; this corresponds to the number of data set inputs (for the Iris classification, this value is 4)
Hidden neurons	Number of hidden neurons, which actually learn the structure of the inputs.
Output neurons	Number of neurons in the output layer, which corresponds to the number of outputs in the data set (for Iris classification, this value is 3)
Use Bias Neuron	This option determines whether all neurons should use a single constant input or the so called bias, which improves learning. This option is usually used, so it is checked by default.
Connect input to output neurons	This option determines whether the input neurons should be directly connected to the output neurons
Transfer function	The transfer function that will be used by the hidden and output layers. Commonly used ones are Sigmoid and Tanh.
Learning rule	The specific type of the Backpropagation algorithm that will be used by the network. Neuroph supports several types: Backpropagation, Backpropagation with Momentum, Resilient Propagation. The Backpropagation with Momentum algorithm is used in this example for Iris classification problem, since the additional momentum parameter improves learning significantly.

Tab III *Multi Layer Perceptron wizard parameters*

The Multi Layer Perceptron network created by the wizard is automatically opened in the visual editor window, with the component palette available on the right, for additional inspection and editing (Fig. 25). The additional neuron in the input and hidden layers is the bias neuron with constant output.

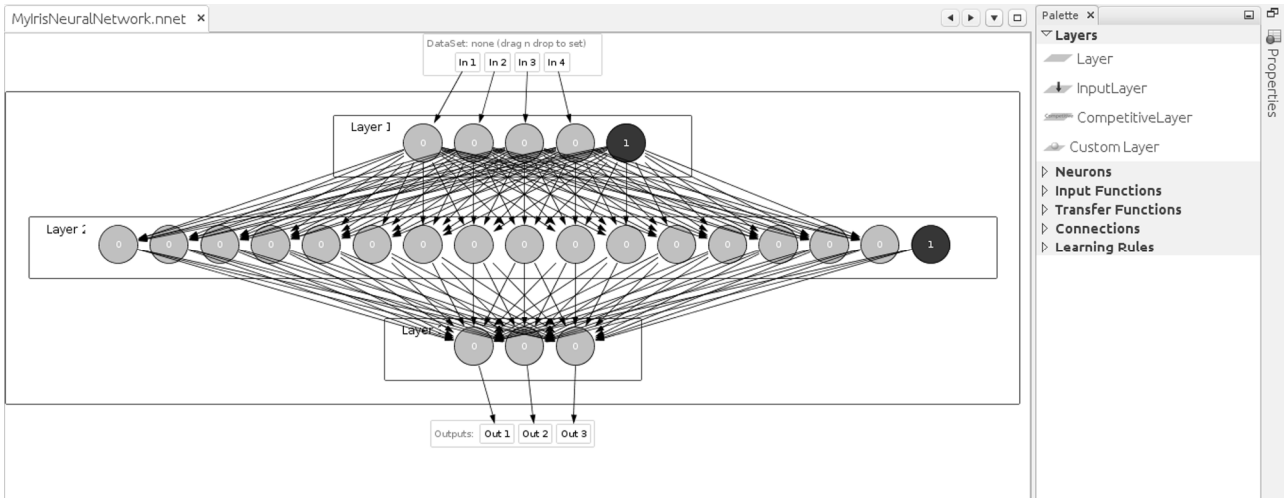


Fig. 25 *Multi Layer Perceptron created with the New Neural Network wizard*

The properties of each neural network component can be inspected in the Explorer and properties windows , and new layers, neurons and connections can be added or changed using the visual editor and the component palette.

Step 4. Basic neural network training

To train the created Multi Layer Perceptron with the data set imported in the second step, the procedure is the same as in the previous example:

1. Drag 'n' drop the data set to the specified area in the neural network window, and click the *'Train'* button in the toolbar (Fig. 26)

2. Set the training parameters in the dialog and click the *'Train'* button (Fig. 11)

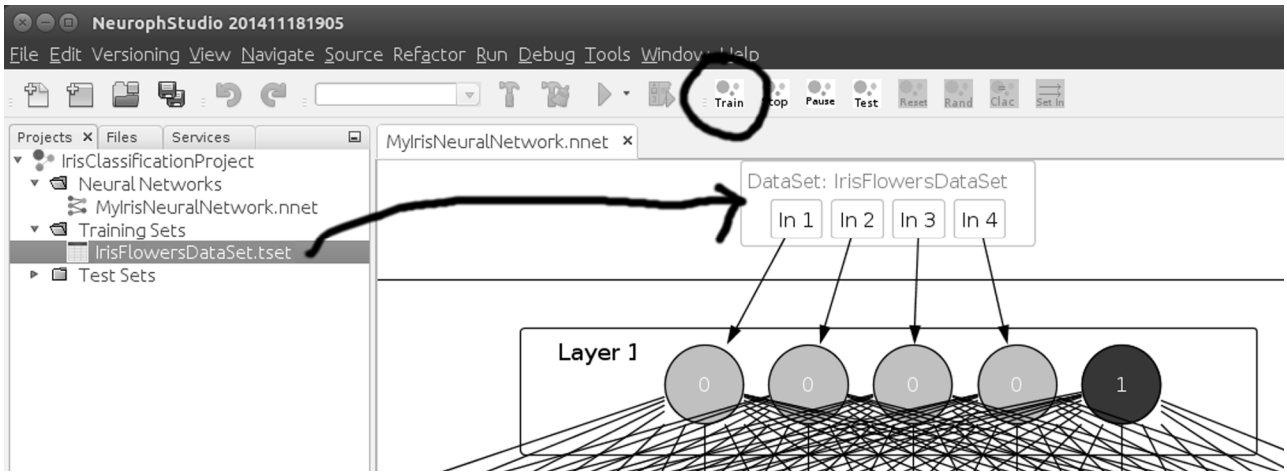


Fig. 26 *Starting the neural network training for the specified data set using drag 'n' drop*

3. During the training, Neuroph Studio displays the real-time Total Error graph, as in the previous example (Fig. 12).

4. After the training, Neuroph Studio displays the Total Network graph for the entire training. If the training does not converge, it can be stopped at any time.

After the training is completed, the user can randomize and modify the existing neural network and repeat the training with different training settings, in order to compare the results (e.g. the number of training iterations). The user can also create new neural networks with different architectures, or even different types of neural networks within the same project. Also it is very easy to execute the basic operations on the data set such as normalization, shuffling, and sub-sampling (by right clicking data set in the project window, and selecting appropriate option from right click menu).

So, the user can easily create different subsets from the original data set , which can be used for training and evaluating neural networks (this procedure can also be automated, which is explained in a subsequent section). This way, Neuroph Studio provides an easy to use, visual environment for experimenting with different neural network settings and types.
After the network has been trained, the next step is the evaluation of how good the network is trained, or how good it solves the problem it is designed for – in this case the Iris flower classification.

Step 5. Evaluating a neural network classifier

Neuroph provides a number of classifier performance measures, which can be used for evaluating neural network classifiers created in Neuroph. It can create a confusion matrix for a given neural network and data set , and then calculate the following classifier performance measures: accuracy, precision, recall, sensitivity, specificity, false positive rate, false negative rate, error rate, false discovery rate, F-measure, Matthews Correlation Coefficient, Q9, and balanced classification rate. It is important to note that Neuroph can provide these measures both for binary and multi-class classification, in which case it calculates all of these measures for each class (many-to-one approach). All of these measures are available on a single click in the Neuroph Studio GUI.

The Iris flowers classifier neural network created in this example can be evaluated in three steps:

1. Label output neurons of the neural network with class names (these labels will be used by the evaluator as the class names)

2. Drag 'n' drop the data set to use for evaluation into the corresponding area in the neural network window

3. Neural network classifier evaluation is launched from the main menu: *'Main Menu > Tools > Classifier Test' , which w*ill open the window with classifier performance measures (Fig. 27)

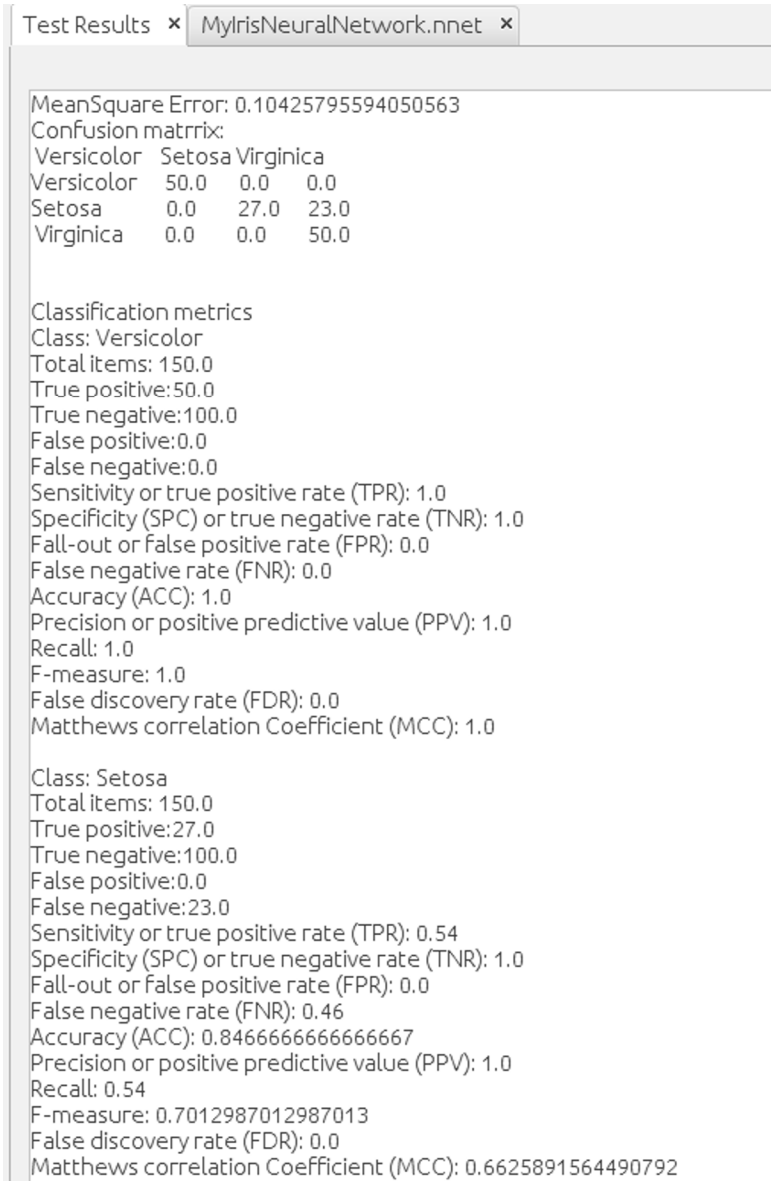


Fig. 27 *Performance evaluation of the Iris neural-network-based classifier*

Figure 27. shows the confusion matrix and classification performance measures for the two classes.

Confusion matrix indicates that all samples of the linearly separable classes (Versicolor and Virginica) are classified correctly, whereas 27 samples of Setosa are classified correctly, and 23 samples incorrectly. All of the supported classification metrics are calculated and shown for all classes.

This result is not satisfactory, which indicates that some different architecture and training settings should be used in order to get better results. For example, if Multi Layer Perceptron with three hidden layers is used with 10, 8 and 6 neurons in each hidden layer respectively, and 0.005 learning rate and 0.3 momentum are used in training settings, only 5 samples of the Setosa class are classified incorrectly.

Since in all these examples the entire data set is used for training, there is a danger of over-fitting, but Neuroph Studio provides an option for sub-sampling and creating training and test sets from the original data set in the data set right-click menu.

This way, Neuroph and Neuroph Studio GUI provide the operations commonly used in the neural network training, and the users can easily experiment with different settings and approaches. This is very helpful and intuitive for users who are new in the neural network and machine learning world.

Besides this manual procedure, in which the user controls and executes every step, Neuroph Studio also provides an automated K-fold cross-validation procedure as part of the training dialog (Fig. 7). This procedure automatically generates K different pairs of training and test sets, and trains K different neural networks.

Table IV shows the basic usability metrics for the entire procedure of creating Multi Layer Perceptron classifier using GUI. The metrics include: the number of dialogs, total number of required parameters in all dialogs, and additional actions (mouse, toolbar). The metrics in Table IV show that almost all operations required to train a neural network are executed through a series of wizard dialogs, which guide the user through the process and represent a good usability practice.

Basic usability metrics	
Dialogs	5
Required Parameters	17
Actions	2

Tab. IV *Basic usability metrics for training Multi Layer Perceptron classifier using GUI*

3.2.2. Iris Classification Example in Java Code

This section describes how to implement the entire procedure of training neural network for Iris classification problem in Java code, using the Neuroph framework. The procedure includes all the steps described for solving the Iris classification using Neuroph Studio GUI:

1. Neural network training
2. Data set sub-sampling
3. Classifier performance evaluation
4. Cross-validation procedure

Code listing 1. shows how to import the Iris data set from the provided CSV file and train the corresponding Multi Layer Perceptron neural network.

The instance of the data set which can be used for training a neural networks in Neuroph is created from the corresponding file by using the *createFromFile()* method of the *DataSet* class (lines 2 and 3). The method *createFromFile()*takes as its input parameters the file name, the number of inputs, the number of outputs, the value delimiter, and a boolean flag that indicates if the input file contains column names(although all of these parameters are intuitive, for more detailed explanation see table II).

```
1:  // create data set  from csv file
2:  DataSet irisDataSet  =
3:    DataSet.createFromFile("iris_data.txt", 4, 3, "\t", false);
4:
5:  // Create multi layer perceptron neural network
6:  MultiLayerPerceptron neuralNet = new MultiLayerPerceptron(4, 16, 3);
7:
8:  // get learning rule from neural network
9:  Backpropagation learningRule = neuralnet.getLearningRule();
10: // set max error parameter
11: learningRule.setMaxError(0.01);
12: // set learning rate parameter of backpropagation learning rule
13: learningRule.setLearningRate(0.02);
14: // set max numberof learning iterations
15: learningRule.setMaxIterations(10000);
16:
17: // start network training by calling its learn method
18: neuralNet.learn(irisDataSet);
19:
20: // save trained neural network
21: neuralNet.save("irisClassifier.nnet");
```

Listing 1. *Import data set from the CSV file and train the Multi Layer Perceptron*

The instance of the Multi Layer Perceptron neural network is created in Java code using the constructor of the *MultiLayerPerceptron* class (line 6). This constructor takes the numbers of neurons in the layers as input parameters, so this code creates a Multi Layer Perceptron with 4 input, 16 hidden, and 3 output neurons. It uses the Sigmoid transfer function, and standard Backpropagation as its default learning rule. The class *MultiLayerPerceptron* extends the base *NeuralNetwork* class. An instance of the default learning rule is obtained by using a simple getter method (line 9), and learning rule parameters for Backpropagation (max error, learning rate, and max iterations) are set in lines 11, 13, and 15, respectively.

The training procedure is started by invoking the *learn()* method of the *NeuralNetwork* class and providing the data set as the input parameter (line 18). The training procedure will be finished when the total network error drops below the error threshold (the maxError parameter) or the algorithm reaches the maximum number of iterations (the maxIterations parameter). After the learning procedure has finished, the network is saved in a file by using the *save()* method of the *NeuralNetwork* class. This method uses the built-in Java serialization mechanism to save an entire object to a file, and expects the file name as the input parameter.

The result of this code is an instance of Multi Layer Perceptron neural network, trained for the Iris flower classification and saved on disk. Table V shows the basic code metrics for listing 1. These code metrics clearly indicate ease of use of Neuroph for creating an instance and training of Multi Layer Perceptron for classification problems.

Basic code metrics for training Multi Layer Perceptron	
Lines of Code	8
Number of classes	3
Number of method calls	8
Number of method parameters	11

Tab. V *Basic code metrics for training the Iris classifier*

Other types of transfer functions can be used by providing transfer function type to the *MultiLayerPerceptron* constructor. Different types of learning rule can be used by setting different learning rule using the setter method of the NeuralNetwork class. For example, to use Resilient propagation, that would be *neuralNet.setLearningRule(new ResilientPropagation())*.To use different neural network architecture, which means different number of layers and different numbers of neurons in each layer, just provide the desired values for these to the *MultiLayerPerceptron* constructor. For example, to apply this procedure for a different data set , appropriate parameters should be provided to *DataSet.createFromFile()*method, and other parameters (like the neural network architecture and learning rule parameters) should be set.

The procedure described in listing 1. trains the network, but most likely it will do over-fitting since the entire data set is used. Sub sampling of the original data set in order to create the training and test sets is a commonly used operation, and the *DataSet* class provides methods for this operations. It also supports some other typical preprocessing methods, like shuffling and normalization.

Code listing 2. shows several ways to create training and test sets from the original data set .

```
1:  // create data set  from csv file
2:  DataSet irisDataSet  =
3:        DataSet.createFromFile("iris_data_normalised.txt", 4, 3, "\t", false);
4:
5:  // shuffle the original data set
6:  irisDataSet.shuffle();
7:  irisDataSet.normalize(new MaxNormalizer());
8:
9:  // Create training and test set pair
10: DataSet [] trainAndTestSets= irisDataSet
11: .createTrainingAndTestSubsets(60,40);
12: List<DataSet> subsets = irisDataSet.sample(new SubSampling(60, 40));
13: List<DataSet> subsets = irisDataSet.sample(new SubSampling(60, 20, 20));
    List<DataSet> subsets = irisDataSet.sample(new SubSampling(4));
```

Listing 2. *Create training and test sets in Java code*

After the data set is created from the file (line 2), the order of data set rows can be randomized using the *shuffle()* method (line 6), and the entire data set can be normalized using the *normalize()* method (line 7), which takes some implementation of the *Normalizer* interface to be provided as the input parameter. Neuroph provides several typical normalization methods, like max normalization (*MaxNormalizer*), max min normalization (*MaxMinNormalizer*), range normalization (*RangeNormalizer*) and decimal scale normalization (*DecimalScaleNormalizer*).

The basic way to create the training and test sets is to use the *createTrainingAndTestSubsets()* method of the *Data set* class (line 10). This method takes as the input parameter the ratio of training and test sets in percents, and returns an array with training and test sets.

The *DataSet* class also provides the *sample()* method, which takes an implementation of the *Sampling* interface as the input parameter. This example is using instances of the *SubSampling* class (lines 11, 12, 13), which provides a method for selecting a subset of the given data set . An instance of the *SubSampling* class is created using the constructor that can take ratios of subsets (lines 11, 12), or number of subsets (line 13) to extract. The *sample()*method returns a list of generated subsets. Custom sampling methods can be created by implementing the *Sampling* interface.

Code listing 3. shows how to run performance evaluation on a trained neural network classifier and the provided Iris data set .

```
1:  // load trained network from file
2:  MultiLayerPerceptron neuralNet = (MultiLayerPerceptron)
3:  NeuralNetwork.createFromFile("irisClassifier.nnet");
4:  // import data set  from file
5:  DataSet dataSet  =
6:         DataSet.createFromFile("iris_data_normalised.txt", 4, 3, ",", false);
7:  // create an array of class names (needed for classification evaluator)
8:  String[] classNames = {"Virginica", "Setosa", "Versicolor"};
9:  // create an instance of evaluation procedure
10: Evaluation evaluation = new Evaluation();
11: // add MSE and classification evaluators
12: evaluation.addEvaluator(new ErrorEvaluator(new MeanSquaredError()));
13: evaluation.addEvaluator(new ClassifierEvaluator.MultiClass(classNames));
14: // run evaluaton for specified neural network and data set
15: evaluation.evaluateDataSet (neuralNet, data set );
16:
17: // get classification evaluator
18: ClassifierEvaluator evaluator =
19: evaluation.getEvaluator(ClassifierEvaluator.MultiClass.class);
20: // get and print confusion matrix generated by evaluator
21: ConfusionMatrix confusionMatrix = evaluator.getResult();
22: System.out.println("Confusion matrrix:\r\n");
23: System.out.println(confusionMatrix.toString() + "\r\n\r\n");
24:
25: // calculate classification metrics from confusion matrix
26: ClassificationMetrics[] metrics =
27: ClassificationMetrics.createFromMatrix(confusionMatrix);
28: ClassificationMetrics.Stats average =
29: ClassificationMetrics.average(metrics);
30:
31: // print classification metrics
32: System.out.println("Classification metrics\r\n");
33: for (ClassificationMetrics cm : metrics) {
34:     System.out.println(cm.toString() + "\r\n");
35: }
36: System.out.println(average.toString());
```

Listing 3. *Evaluating neural network classifier performance, in Java code*

The code from listing 3 loads the trained neural network from the appropriate file (line 2), imports the corresponding data set from its file (line 5), and calculates the mean squared error and classification metrics for the given neural network and data set. The main classes used for classifier evaluation are:

Evaluation, ClassifierEvaluator, ConfusionMatrix,and ClassificationMetrics.*Evaluation* represents the general evaluation procedure, which consists of running a collection of evaluators (lines 12 and 13) on a specified neural network and data set (line 15). The evaluators can be dynamically added to the evaluator, depending of the type of the problem being evaluated. In this case,the *MeanSquaredError* (line 12) and *ClassifierPerformance* (line 13) evaluators are used. The *ClassifierEvaluator* class tests a neural network classifier for all data in a given data set , and generates confusion matrix that can be obtained using *Evaluator*'s *getResult()* method (line 21).The *ConfusionMatrix* class holds all information about the generated confusion matrix, class names, matrix values, and methods for getting the basic metrics (true positive, true negative, false positive and false negative). The *ClassificationMetrics* class provides methods for calculating a number of classification metrics based on the basic metrics from *ConfucionMatrix* (lines 26 and 27). The supported classification metrics are listed in section 3.2.1. under step 5, and the output of the code is shown in figure 27. Table VI shows the basic code metrics for code listing 3 (printing lines, network and data set loading are ignored). These metrics indicate the simplicity and ease of use of Neuroph for classifier performance evaluation and cross-validation in Java code.

Basic code metrics for evaluating a neural network classifier	
Lines of Code	9
Number of classes	7
Number of method calls	11
Number of method parameters	9

Tab. VI *Basic code metrics for evaluating Neuroph-based Iris classifier*

Basic code metrics for cross-validation	
Lines of Code	5
Number of classes	3
Number of method calls	5
Number of method parameters	5

Tab. VII *Basic code metrics for cross-validation with Neuroph*

Code listing 4. shows how to run the cross-validation procedure on a trained neural network and provided data set .

```
1:  // load saved trained network from file
2:  MultiLayerPerceptron neuralNet = (MultiLayerPerceptron)
3:  NeuralNetwork.createFromFile("irisNet.nnet");
4:
5:  // create data set  from file
6:  DataSet data set  =
7:   DataSet.createFromFile("data_sets/iris_data_normalised.txt", 4, 3, ",");
8:  // class names for classifier evaluator
9:   String[] classNames = {"Virginica", "Setosa", "Versicolor"};
10:
11: // create crossvalidation procedure for given neural network and data set
12: CrossValidation crossval = new CrossValidation(neuralNet, data set , 5);
13: // add classifier performance evaluator
14: crossval.addEvaluator(new ClassifierEvaluator.MultiClass(classNames));
15: // run  crossvalidation
16: crossval.run();
17: // get crossvalidation results
18: CrossValidationResult results = crossval.getResult();
19:
20: // print crossvalidation results
21: System.out.println(results);
```

Listing 4. *Running cross-validation procedure, in Java code*

The code from listing 4. loads the trained neural network from its file (line 2), imports the corresponding data set from another file (line 6), and runs an appropriate cross-validation procedure for a given neural network and the corresponding data set (lines 12-16).

The *CrossValidation* class contains all elements for running the cross-validation procedure: the neural network, the corresponding data set and the evaluators used in the validation procedure. It creates the specified number of subsets of the original data set , then trains the neural network with one subset, and uses rest of the subsets for evaluation. It repeats this procedure for each generated subset. This way, it automates the cross-validation procedure. It can be configured to use various evaluators and data sampling techniques. The *CrossValidationResult* class (line 18) holds the results from all cross-validation iterations. Table VII shows the basic code metrics for listing 4 (printing lines, network and data set loading are ignored).

3.3. Image Recognition

This example shows how to use Neuroph for basic image recognition. It also shows how Neuroph can be extended and used for other specific application domains - how its API can be extended, and how all that can be supported with specialized tools.
Image recognition in Neuroph is based on using raw pixel color information. RGB color information is extracted for each image pixel, and is used to create an input vector, which is then used for neural network training. This way, each input image corresponds to a single input vector, and the neural network task is to learn the mappings between the input color vectors and a set of predefined image labels. It is also possible to map a set of input color vectors to the same label, so this problem is very similar to the classification problem.
Neuroph provides a specialized wizard to create data set s from images, and the Multi Layer Perceptron neural network with image recognition functions.
Through a series of dialogs, the wizard requires from the user to provide the images to recognize and some additional settings. The image recognition wizard is launched from the main menu:

Main menu > File > New > Neuroph > Image recognition

In the first step, the wizard asks user to select the images for recognition (Fig. 28).

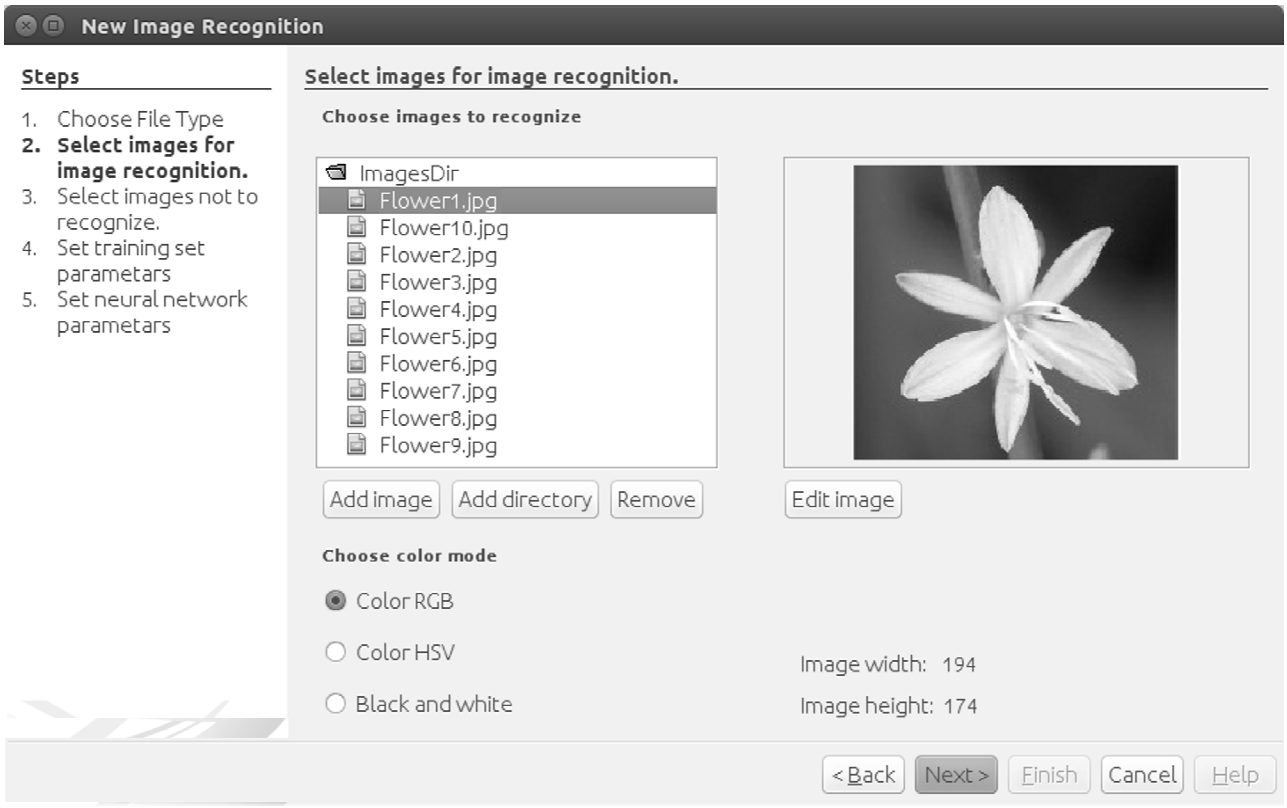


Fig. 28 The *Image recognition wizard, step 1 - image selection*

In second step (Fig. 29),the wizard asks for the data set name and the sampling resolution. The sampling resolution represents dimensions to which the images selected for recognition will be scaled. The smaller dimensions, the less the number of neurons, and the faster the learning. Too small

dimensions can have negative impact on the recognition accuracy, so the right values are determined experimentally.

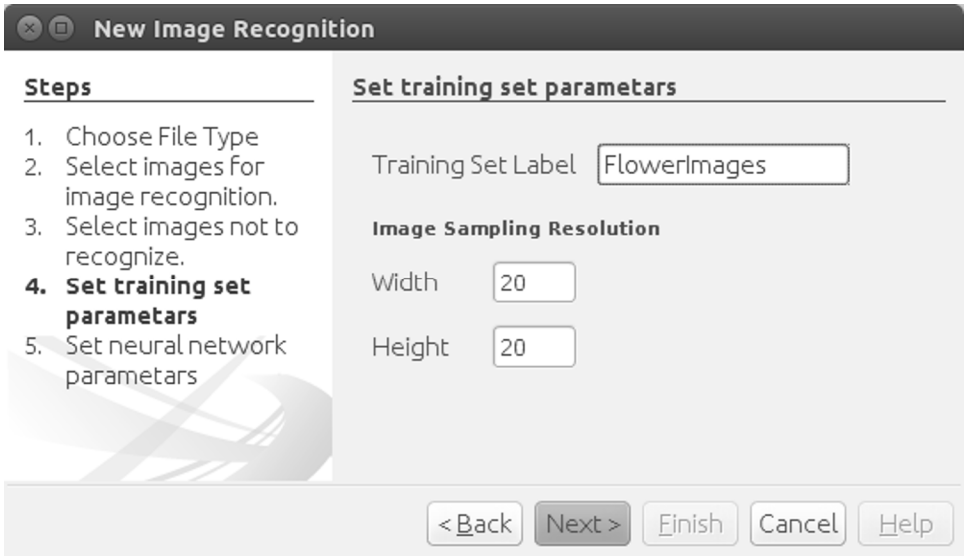


Fig. 29 The *Image recognition wizard step 2 - data set settings*

In the third step (Fig. 30),the wizard asks for basic Multi Layer Perceptron neural network settings like the transfer function and the number of hidden layers (already explained in detail in the classification example).

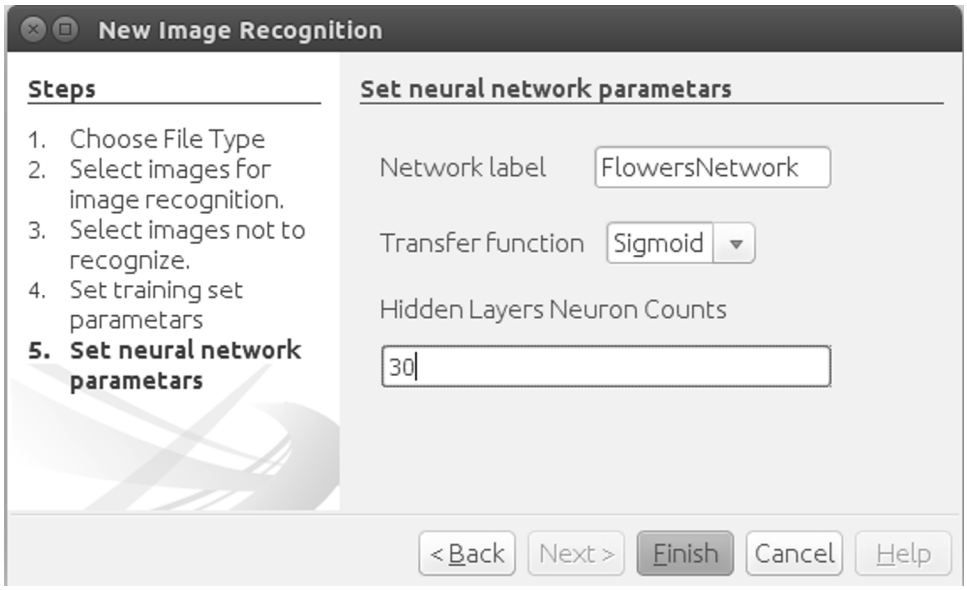


Fig. 30 The *Image recognition wizard step 3 - neural network settings*

When the wizard is finished, it automatically creates the data set from the provided images and the Multi Layer Perceptron neural network with Image Recognition plug-in, which provides simple Java API for image recognition. The network is trained using the procedure described in the classification example: '*drag 'n' drop'* the data set to the network, and click the '*Train'* button in the toolbar.
The trained network can be tested in the '*Image Recognition Test'* window, where the user can select an

image and see the network output for each image label, which indicates that a specific image is recognized. The image label/output neuron with the highest activation is the one that is recognized (Fig. 31).

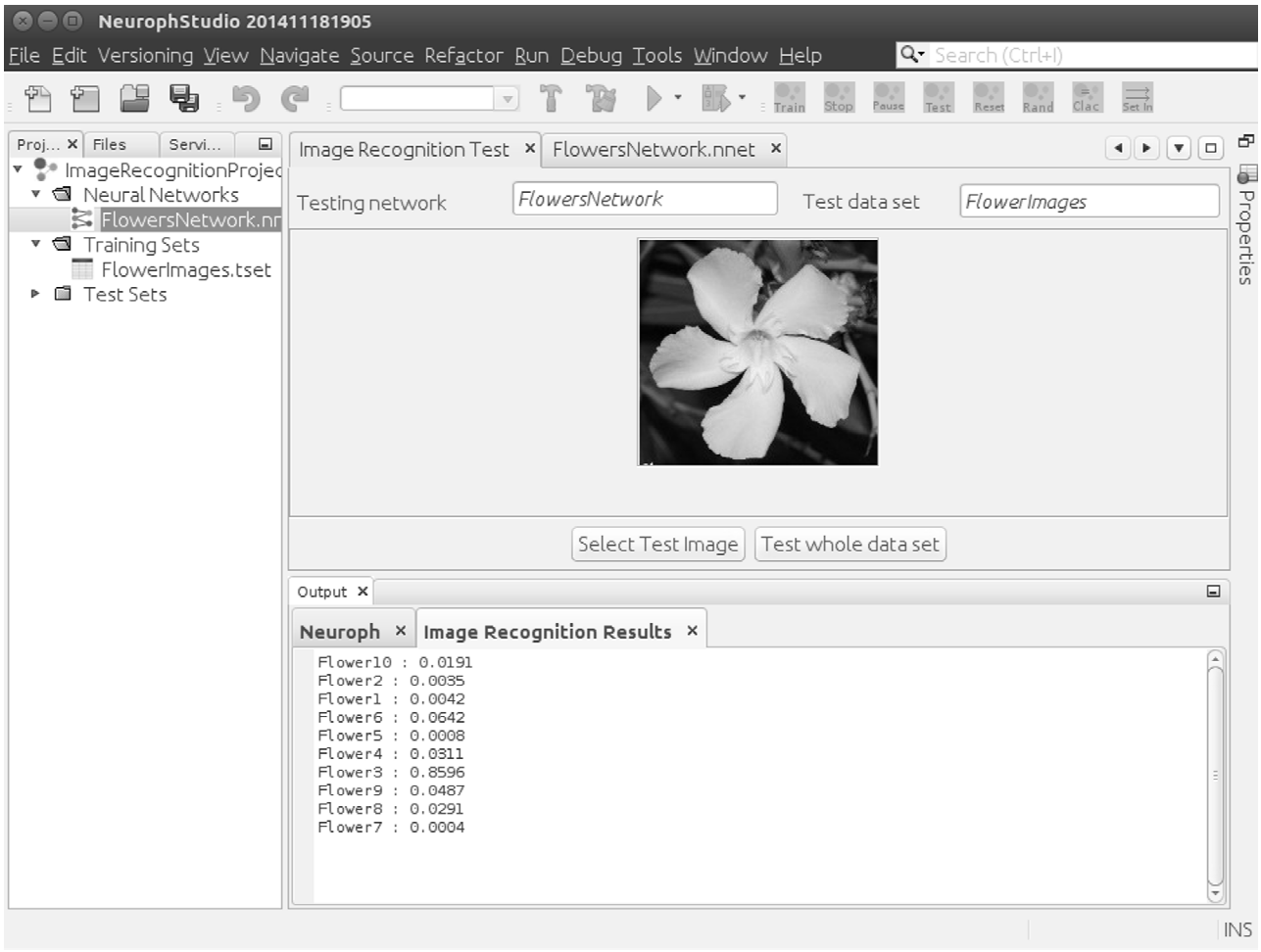


Fig. 31 *Image recognition testing*

Table VIII shows the basic usability metrics for the entire procedure of creating Multi Layer Perceptron for image recognition using GUI. The metrics include: number of dialogs, total number of required parameters in all dialogs and additional actions (mouse, toolbar). The metrics in Table VIII show that almost all operations required to train neural network are executed through a series of wizard dialogs, which guide the user through the process and represent a good usability practice.

Basic usability metrics	
Dialogs	6
Parameters	13
Actions	2

Table VIII *Basic usability metrics for training Multi Layer Perceptron for image recognition using GUI*

The created image recognition neural network can be saved as a serialized Java component, and used in other Java applications.
The code in listing 5 shows how to use the image recognition neural network created with this wizard in other Java applications.
In line 2, the neural network created in Neuroph Studio is loaded from a file, and in line 5 an instance of the image recognition plug-in is acquired by using the getter method *getPlugin()*.The actual recognition is run in line 10, using the method *recognizeImage()* from the *ImageRecognitionPlugin* instance. This method takes an image or an image file that is processed, and returns a hash map with the image labels (names) and the likelihoods that a specific image is recognized. So, besides the neural network loading and exception handling, the actual image recognition is accomplished in just two lines of code. In the entire sample, only two classes from Neuroph are used – *NeuralNetwork* and *ImageRecognitionPlugin*.

```
1:  // load trained neural network saved with NeurophStudio
2:  NeuralNetwork nnet= NeuralNetwork.createFromFile("MyImageRecognition.nnet");
3:
4:  // get the image recognition plugin from neural network
5:  ImageRecognitionPlugin imageRecognition =
6:  (ImageRecognitionPlugin)nnet.getPlugin(ImageRecognitionPlugin.class);
7:
8:  try {
9:   // actual call to image recognition method
10:    HashMap<String, Double> output =
11:                imageRecognition.recognizeImage(new File("someImage.jpg"));
12:    System.out.println(output.toString());
13: } catch(IOException ioe) {
14:    System.out.println("Error: could not read file!");
15: } catch (VectorSizeMismatchException vsme) {
16:    System.out.println("Error: Image dimensions dont match !");
17: }
```

Listing 5. *Image recognition in Java code, with the network created in Neuroph Studio*

Table IX shows the basic code metrics for using image recognition neural network in Java code, which clearly indicates ease of use of Neuroph for this type of applications.

Basic code metrics for image recognition	
Lines of Code	2
Number of Neuroph classes	2
Number of method calls	2
Number of method parameters	2

Tab. IX *Basic Java code metrics for image recognition with Neuroph*

It is also important to outline the flexibility of the code shown above with respect to using different types of neural networks. In order to do that, nothing has to be changed in the code since the same *ImageRecognitionPlugin* is used and it represents an interface for this specific domain.

This is also a good example of how Neuroph can, and should be extended for other application-specific domains using its plugin-based architecture. More technical details about it are explained in the next section.

4. Extending the Neuroph framework

This section provides a brief overview of the classes and methods that represent the main extension points. It outlines the required steps and provides guidelines for creating new components in the Neuroph framework. Typical scenarios for extensions include:

1. Creating a new type of neural network architecture
2. Creating a new type of learning rule
3. Creating domain-specific operations
4. Creating a new type of neuron layers
5. Creating a new type of neurons
6. Creating a new type of input function
7. Creating a new type of transfer function

The overall architecture of Neuroph facilitates maximum reusability for extensions by implementing the general structure and logic in the base classes, so new specific features can be added in the extended classes by overriding abstract methods or providing implementations of some interfaces.

Creating a new type of neural network architecture

Create a new class that extends class *NeuralNetwork*.

Add the method *createNetwork()* that creates layers of neurons and sets a learning rule. Some specific types of neural networks might require new types of learning algorithms. The recommended practice is to provide a constructor or an inner builder class to create a network instance, and setter methods that allow setting of various network parameters.

Examples: MultiLayerPerceptron, ConvolutionalNetwork, HebbianNetwork

Creating a new type of learning algorithm

Create a new class thatextends the abstract class *LearningRule* or some of its subclasses (*IterativeLearning*, *Supervised*, *Unsupervised*) and implement the corresponding abstract method:

LearningRule
    abstract public void learn(DataSet trainingSet)

IterativeLearning extends LearningRule
    abstract public void doLearningEpoch(DataSet trainingSet)

Supervised extends IterativeLearning
abstract protected void updateNetworkWeights(double[] outputError)

Unsupervised extends IterativeLearning
abstract protected void updateNetworkWeights()

The *LearningRule* class is the most general base class for learning rules, whereas *IterativeLearning* is the base class for iterative learning procedure and it provides a general iterative learning procedure. The *SupervisedLearning* class is the base class for the family of supervised learning algorithms, and its subclasses should implement a method for updating the network weights, based on the output error.

Examples: *LMS, Backpropagation, RBFLearning*

UnsupervisedLearning is the base class for the family of unsupervised learning algorithms, and its subclasses should implement a method for updating the network weights.

Examples: *KohonenLearning, HebbianLearning, CompetitiveLearning*

Creating domain specific operations (API)

Domain specific operations (like image recognition) should be created by using the Neuroph plugin system. To create a new plug in, create a new class that extends *org.neuroph.core.PluginBase*and add the required methods, which are usually related to setting the network input and interpreting the network output for a specific application. Plug-ins are added to a neural network using the *addPlugin()* method, and requested using the *getPlugin()* method.

Examples: *ImageRecognitionPlugin, OcrPlugin*

Creating a new type of neuron layers
Create a new class that extends the class *org.neuroph.core.Layer* and optionally override some of its methods, usually:

public void calculate()

Since the *Layer* class is the container for neurons, and its default implementation of the *calculate()* method does not need to be changed for most purposes, the newly extended class usually adds more features to the existing basic layer type.

Examples: *Layer2D, ConvolutionalLayer, CompetitiveLayer*

Creating a new type of neuron

Create a new class that extends the class *org.neuroph.core.Neuron* and override the method *public void calculate().* This method calculates the neuron's output. Its default implementation in the *Neuron* class calculates the value of the neuron's input function and feeds that value into the transfer function. The output of the transfer function is the neuron's output.

Different input and output functions can be easily changed using the neuron's setter methods, and the only situation when a different type of neuron is needed is when a different method of computation is required.

Examples: *CompetitiveNeuron, InputNeuron, BiasNeuron*

Creating a new type of input function:
Create a new class that extends the abstract class *org.neuroph.core.input.InputFunction* and implement its abstract method:

abstract public double getOutput(Connection[] inputConnections);

This method takes an array of a neuron's input connections, and its implementations should calculate and return the result of the corresponding input function.

Example: *org.neuroph.core.input.WeightedSum*

Creating a new type of transfer function

Create a class that extends the abstract class *org.neuroph.core.transfer.TransferFunction.*Implement its abstract method that calculates and returns the value of the transfer function:

abstract public double getOutput(double net);

Override its method which returns the first derivative of the transfer function:

public double getDerivative(double net)

Example: *org.neuroph.core.transfer.Sigmoid*

5. Conclusion

This paper describes Neuroph, an open-source neural network development environment written in Java. The application cases presented in the paper show how to use Neuroph to:

1. Demonstrate the basic neural network concepts, visualize learning, network architecture, and data set s.
2. Solve a classification problem using the Multi Layer Perceptron neural network, both with GUI and in Java code (where the same procedure is applicable for other types of problems and neural networks)
3. Do basic image recognition, and customize Neuroph for some specific application domains.

These examples also demonstrate the most important Neuroph features, which are:

- It is easy to learn and use, thanks to a small number of well-designed classes and the development environment that provides wizard-based and visual tools.
- Flexible and extensible design, with clear, well-defined and comprehensive extension points, which make it easy to develop new types of learning algorithms and neural network architectures.
- High reusability, since it is easy to deploy in different environments (other Java applications), and the existing code facilitates development of new components/extensions.

Neuroph is intended to be used by students, teachers, researchers and software developers interested in using neural networks.
Future development will include new types of neural network architectures, learning algorithms, and automated tools to support typical workflows (training, validation, testing, deployment) when working with neural networks.

Bibliography

1. **Gosling, James.** *The Java Language Environment.* s.l. : Sun MicroSystems, 1995.
2. **NetBeans Project.** https://netbeans.org/features/platform/. *NetBeans Platform.* [Online] Oracle, 9 3, 2015. [Cited: 9 5, 2015.] https://netbeans.org/features/platform/.
3. *Java tools for research and education in artificial neural networks.* **Corbett, F. D., Card, H. C.** Waterloo : IEEE, 1998. IEEE Canadian Conference on Electrical and Computer Engineering. pp. 417-420. DOI: 10.1109/CCECE.1998.682773.
4. **Beckenkamp, F. G.** A component architecture for artificial neural network systems. Constance : PhD thesis, University of Constance, 2002.
5. *A GUI-based artificial neural network simulator.* **Nasr, G.E., Joun, C., Zaatar, W.** Belgrade : IEEE, 2004. 7th Seminar on Neural Network Applications in Electrical Engineering. pp. 135-138. DOI: 10.1109/NEUREL.2004.1416556.
6. **Mackin, K. J.** Modeling layered artificial neural networks using a visual programming paradigm. *14th International Symposium on Artificial Life and Robotics.* Oita, Japan : s.n., 2009. pp. 422-424. DOI: 10.1007/s10015-009-0701-2.
7. *A survey of artificial neural network training tools.* **Baptista, D. , Morgado-Dias, F.** 3-4, London : Springer, 2013, Neural Computing and Applications, Vol. 23, pp. 609-615. ISSN 0941-0643, DOI: 10.1007/s00521-013-1408-9.
8. *JLCNN: An object-oriented Java package for low complexity neural networks.* **Dogaru, I., Dogaru, R.** 2013 . 4th International Symposium on Electrical and Electronics Engineering (ISEEE). pp. 1-6. DOI: 10.1109/ISEEE.2013.6674317.
9. *Encog: Library of Interchangeable Machine Learning Models for Java and C#.* **Heaton, Jeff.** 16, 2015, Journal of Machine Learning Research, pp. 1243-1247.
10. **Marrone, Paolo.** *Java Object Oriented Neural Engine JOONE.* [Online] [Cited: 8 5, 2015.] http://sourceforge.net/projects/joone/.
11. **Zell, Andreas.** *Java Neural Network Simulator JNNS.* [Online] [Cited: 8 7, 2015.] http://www.ra.cs.uni-tuebingen.de/software/JavaNNS/.
12. **Neuroph project.** Contributors. *Official Neuroph site.* [Online] 10 3, 2008. [Cited: 9 10, 2015.] http://neuroph.sourceforge.net/contributors.html.
13. **Eric, Evans, Fowler, Martin.** *Domain-Driven Design: Tackling Complexity in the Heart of Software.* s.l. : Prentice Hall, 2003. ISBN: 978-0321125217.
14. **Pree, Wolfgang.** *Framework Patterns.* New York : SIGS Books, 1996. ISBN:1884842542.
15. —. Object-Oriented Design Patterns and Hot Spot Cards. *IEEE International Conference on the Engineering of Complex Computer Systems (ICECCS97).* Como, Italy : s.n., 1997.
16. **Mohamed E. Fayad, Douglas C. Schmidt , Ralph E. Johnson.** *Building Application Frameworks: Object- Oriented Foundations of Framework Design.* New York : Wiley & Sons, 1999. ISBN 0-471-24875-4 .
17. **Bernard Widrow, M.A. Lehr.** Perceptrons, Adalines, and Backpropagation. [book auth.] Michael A. Arbib. *Handbook of Brain Theory and Neural Networks.* s.l. : MIT Press, 1995.
18. **Hopfield, J. J.** Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences of the USA.* 1982. Vol. 79, 8, pp. 2554-2558.
19. *Bidirectional Associative Memory.* **Kosko, Bart.** 1, Piscataway, NJ, USA : IEEE Press, 1988, IEEE Transactions on Systems, Man and Cybernetics, Vol. 18, pp. 49-60. DOI:10.1109/21.87054.
20. *Self-organized formation of topologically correct feature maps.* **Kohonen, Teuvo.** 1, s.l. : Springer-Verlag, 1982, Biological Cybernetics, Vol. 43, pp. 59-69. DOI:10.1007/BF00337288.
21. *Adaptive pattern classification and universal recoding: I Parallel development and coding of neural feature detectors.* **Grossberg, Stephen.** 3, s.l. : Springer Berlin Heidelberg, 1976, Biological

Cybernetics, Vol. 23, pp. 121-134. ISSN:0340-1200.
22. *Feature discovery by competitive learning.* **Rumelhart, David E., and David Zipser.** 1, s.l. : Elsevier, 1985, Cognitive science , Vol. 9, pp. 75-112. DOI:10.1016/S0364-0213(85)80010-0.
23. **Broomhead, D. S. and Lowe, David.** *Radial basis functions, multi-variable functional interpolation and adaptive networks.* Malvern, United Kingdom : Royal Signals and Radar Establishment, 1988. Technical report, 4148.
24. *Adaptive Neuro-Fuzzy Pedagogical Recommender.* **Zoran Ševarac, Vladan Devedžić, Jelena Jovanović.** 10, s.l. : Elsevier, 2012, Expert Systems With Applications, Vol. 39. ISSN 0957-4174, DOI: http://dx.doi.org/10.1016/j.eswa.2012.02.174.
25. **Donald, Hebb.** *The Organization of Behavior: A Neuropsychological Theory.* s.l. : Psychology Press, 2002. ISBN:978-0805843002.
26. *Deep learning.* **Yann LeCun, Yoshua Bengio, Geoffrey Hinton.** 521, s.l. : Nature Publishing Group, 2015, Nature, pp. 436-444. DOI:10.1038/nature14539.
27. *A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm.* **Martin Riedmiller, Heinrich Braun.** s.l. : IEEE, 1993. IEEE International Conference On Neural Networks. pp. 586-591.
28. **Dieter Jaeger, Ranu Jung.** *Encyclopedia of Computational Neuroscience.* New York : Springer New York, 2015. ISBN 978-1-4614-7320-6.
29. *Simplified neuron model as a principal component analyzer.* **Erkki, Oja.** 3, s.l. : Springer-Verlag, 1982, Journal of Mathematical Biology, Vol. 15, pp. 267-273. DOI 10.1007/BF00275687. ISSN 0303-6812.