COBOSLAB

Cognitive Bodyspaces: Learning and Behavior

Technical Report No. CoboslabY2010N002
April 3, 2010

Echo-State Network In Java: Introduction, Manual, and Evaluations

Johannes Lohmann & Martin V. Butz

COBOSLAB, Department of Psychology
University of Würzburg
Röntgenring 11
97070 Würzburg, Germany
http://www.coboslab.psychologie.uni-wuerzburg.de

Echo-State Network In Java: Introduction, Manual, and Evaluations

Johannes Lohmann* Martin V. Butz†

Abstract

Echo-state networks (ESNs) are a promising approach in training recurrent neural networks (RNNs). This architecture, developed by Jaeger (Jaeger, 2001), avoids most of the common problems associated with the training of RNNs. In this report a Java-based implementation is described that can be used to illustrate the functions of ESNs. Examples will be given and important features of this type of network will be discussed. Furthermore an approach to achieve more stable solutions is introduced, involving the optimization of the connectivity of the dynamic reservoir. Two Appendices provide information about the internal structure of the software. We hope that our implementation can serve as illustration for the possibilities of this flexible and elegant approach.

1 Introduction

Based on the hypothesis of Hebbian Learning as a possible explanation for neural plasticity (i.e. learning), neural networks developed to a standard model for information processing in biological systems (Cruse, 2006). This kind of quantitative models proved to be able to solve a broad variety of complex tasks for example classification, function approximation, data-mining and clustering. One of the most demanding questions remaining concerns dynamic regression problems, also called time series problems. Recurrent neural networks (RNNs) are the typical approach in this case, but the common training procedures, such as back-propagation through time or real-time recurrent learning, face several stability and reliability problems (Jaeger, 2002), which are due to the fact that all connections in the system are adapted:

1. Most of the algorithms that can be utilized to train RNNs are quite slow.

*johannes.lohmann@stud-mail.uni-wuerzburg.de
†butz@psychologie.uni-wuerzburg.de

2. As the connections are adapted simultaneously, chaotic behavior of the systems may occur (bifurcations [1]) and convergence can not be guaranteed.

The echo-state network (ESN) architecture, developed by Jaeger et al. (Jaeger, 2001) avoids many of these known problems (even if convergence still can not be guaranteed). As far as we know, all implementations of ESNs were performed with MATLAB or C++. This article describes a Java-implementation. The formal outlines of training and exploiting ESNs are not explained here. They can be found in various articles from Jaeger et al. (Jaeger, 2001; Jaeger & Haas, 2004), there also exists a scholarpedia article that provides a lot of links for a detailed overview about the mathematics behind ESNs. The main functionality and evaluations in our implementation are based on Jaeger's tutorial article on ESN (Jaeger, 2002). Figure 1 offers an overview of the network structure and the frequently used terms in this report, as well as an illustration of the special aspect of ESNs: Only those weights that determine the connections from the input and the internal layer to the output units are optimized.

The training of ESNs consists of three steps: (1) a washout phase, (2) a sampling phase, and (3) and exploitation phase. At first, the initial random oscillations of the units constituting the dynamic reservoir (the internal layer) have to be extinct. This is done during the *washout phase*, in which the network starts oscillating based on the provided input and output patterns.

Now the main part of the training procedure starts: the *sampling phase*. During this phase the output units receive the dynamic function that should be learned as external (teacher-forced) input. Via the connections represented by the back-projection weights the dynamic reservoir is excited and exhibits a dynamic behavior according to the received input. In the case of batch learning (currently the software does not support online learning) these resulting activations as well as the teacher-forced values of the output units are stored. The data obtained during this phase can now be used to compute appropriate weights for the connections between the output units and the dynamic reservoir. This is a linear regression task. Finally the computed weights are used to calculate an error value (referred to as $\mathrm{MSE}_{sampling}$), which indicates how well the sequence of teacher-forced values was reproduced. This error is a valid pointer for the ability of the network to cope with the current task.

After learning is completed and the connections are set accordingly, the complete system should have developed a stable state that is able to reproduce the desired dynamic. During the last step of the training procedure

[1] Usually similar inputs produce similar outputs. But sometimes it can happen that small changes of the input signal cause qualitative changes in the output. States that induce this kind of behavior are called bifurcation points.

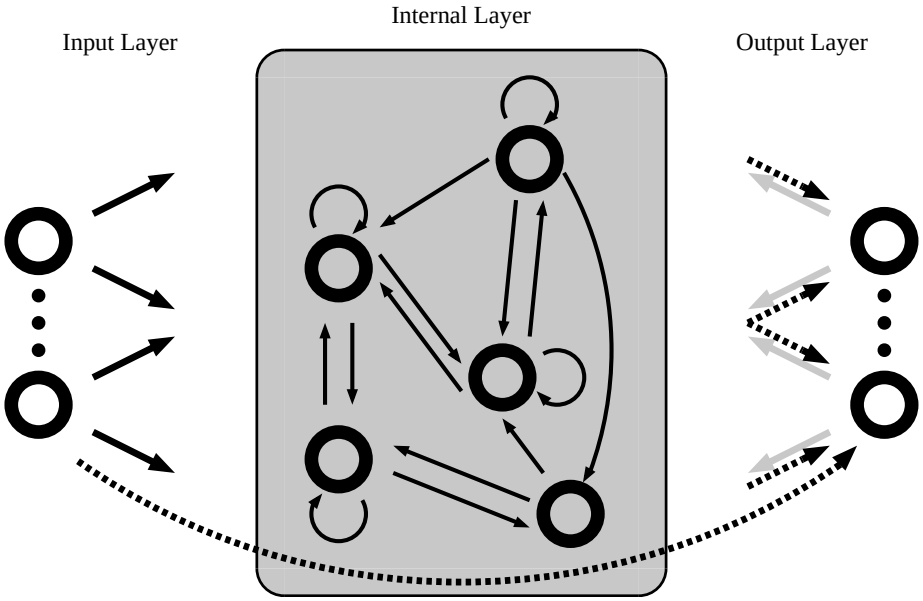

Figure 1: The different layers constituting the network structure. Solid lines indicate fixed connections, dashed lines represent the optimized connections, which are the output weights. The back-projection weights from the output to the internal layer are displayed as gray lines. Please note that by default the internal layer is completely interconnected. However, also sparse connectivity may be selected. For the sake of clarity, not all connections are shown in the graph.

the performance of the network is evaluated, this is called the exploitation phase. A second mean squared error (termed $\mathrm{MSE}_{exploitation}$) is calculated as a measure of the ability of the network to achieve a stable solution for the current task. This is done by comparing the intended output with the actual activations of the output units generated over a given number of time steps.

Optionally input units may be assigned to the network. They can be used to induce a additional dynamic to the units of the dynamic reservoir.

The remainder of the article is structured as follows. The next section provides the information necessary to obtain and run the software. This is followed by a description of the functions of the software, accessible via the graphical user interface. After that a more detailed view on some example problems will be given. Then typical aspects that influence the performance of the network will be discussed. A short discussion concludes the main part of the article. Appendix A provides an overview of the package-structure of the software and Appendix B gives an example for extending the software.

2 Getting Started

This section provides the information necessary to run the software. It is assumed that the code is used via the Eclipse development environment. Alternatively, also the included executable jar file can be executed directly, in which case the online compilation of scripts might not necessarily work, however.

2.1 Prerequisites

At first you have to install the standard Java development kit (JDK), which can be obtained from sun. Please make sure to install all components of the JDK.

Two additional packages are necessary to run the software: The JAMA package and the JUNG package. The software will not work without the JAMA package as all tasks involving linear algebra rely on this package. The functions provided by the JUNG package are optional. They are only used to generate a graph-like visualization of the networks so the code may work without this package. Now the Eclipse IDE should be downloaded and installed. Finally you need the source code that can be obtained from the COBOSLAB download page.

2.2 Setting up the Project

Start Eclipse and create a new Java project. Next, the JAMA and the JUNG packages have to be added to the build path. Right click on the project and select ‘Properties’ from the appearing dialog. Select the ‘Java Build Path’

item and click on the 'Libraries' tab. On the right side there is a button labeled 'Add External JARs...'. Hit this button and mark the jars in the directories where you unzipped the JUNG and JAMA packages to.

In the next step you have to put the source code into the project. Open the directory where you unzipped the `ESNJava1.0` archive. Copy the content from the 'src' directory and paste it into the 'src' directory of the project. Now copy the directories called 'script' and 'DefaultOutput' and paste them into the project. Make sure that you do not paste them into a subdirectory. Finally, all the remaining single files have to be included into the project (again it is important that they are not pasted into a subdirectory).

Now you can run the 'MainGui.java' file in the 'esnMain' package to start the program. The next section describes how the user interface is used that pops up when the program is started successfully.

If you already had a JRE installed before you installed the JDK, it might be the case that scrips will not be executable from within the program since no Java compiler might be found. In this case, choose the JDK folder as the "Alternate JRE" in your "Run Configurations" settings.

3 Graphical User Interface

This section describes the options the user interface provides. Currently the software consists of three main components: The first component realizes the creation and manipulation of ESNs, the preparation of training files is performed by a separate component. The last component is a more or less stand-alone code editor that can be used for scripting issues. This component will be replaced but at the current stage it offers the only possibility for automated data-generation, as the program can't be controlled via setting-files.

The main-functionality is accessible via two menus: The ESN option menu and the training menu.

3.1 The ESN-Option Menu

The items stored in this menu provide basic functions like creating a new ESN from scratch, or loading an existing ESN. Another item offers the possibility to view existing output files (from a single ESN or merged output from different ESNs). The last item allows to access the code editor that might be used to create scripts. The location of the menu and the different items are shown in figure 2.

3.1.1 Create new ESN.

Click this item if you want to create and train a new ESN. At first a frame containing a tree-like object labeled 'Settings' will appear. It is expanded

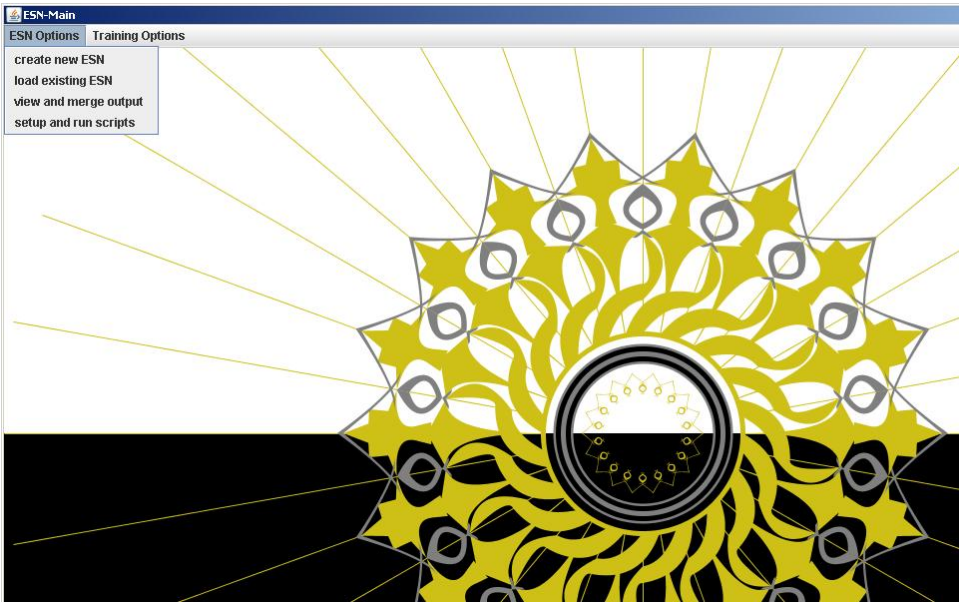

Figure 2: Available items in the ESN-Option Menu.

and contains three branches labeled 'Network', 'IO' and 'Math'. Expand these branches to edit the according properties of the to be created ESN. If you select one of the appearing sub-branches the panel on the right side will display the parameters you may edit. If you drag the cursor over the parameters tool-tips will pop up, providing information about value boundaries and the meaning of the current parameter.

Sub-branches of the 'Network' branch allow to edit the properties of the different layers of the ESN. Options provided by the 'IO' branch allow to set the output directory as well as the input file. Finally the settings available by the sub-branches of the 'Math' branch influence scaling-factors and the noise that is (possibly) applied to the activations of the units of the dynamic reservoir. The example-section covers some of the settings in more detail.

Two buttons are located at the lower right side of the frame. The left one allows to load settings stored in a file. Clicking the other button enables the user to save the current settings.

After all parameters are set, the network can be created by clicking the button labeled 'start' on the bottom right side of the panel. This will open a dialog that ask the user to assign the time steps stored in the selected input file to the different stages of the training process (washout, sampling and exploitation). By default the ratio is 1:2:1 (if your input file contains 300 time steps, the proposition will be 75 time-steps for washout as well as for exploitation and 150 time-steps for sampling). Clicking the 'apply' button will create a new ESN and a new tab (labeled 'ESN') will appear.

This tab contains four to five tabs itself. If you choose to use an input layer, there will be five tabs, four otherwise, one for each layer and one for the mean squared errors. The first tabs contain the different weight matrices (you can not edit the values manually in this view). The last tab contains two error values, one for the sampling phase and one for the exploitation phase. There are four buttons at the bottom of the tab, clicking the 'run training' button will start the sampling phase. If this is finished, the tab labeled 'Output-Layer' contains the estimated weights for the connection between the output units and the dynamic reservoir. Furthermore the mean squared error of the sampling phase is calculated and added to the last tab.

Now the exploitation phase can be initialized by clicking the corresponding button. The results are stored and used to calculate the second mean squared error.

If the user wants to save the data obtained during the training procedure, he has to click the third button. This will print the data into the file selected in the 'IO' settings.

The last button allows you to discard the estimated weights and to restart sampling and exploitation without generating a new network (that is always possible, simply by clicking the 'start' button in the ESN-settings tab), this is only useful if you use noise during sampling, otherwise the results will be identical.

After the sampling and the exploitation phase the 'Visualization Options' menu of the frame is completely accessible. It offers two options: A graph-like visualization of the network structure and a graphical representation of the activations of the different units during the training procedure.

The 'Network-Visualization' item will open a new frame containing a representation of the network as a graph. It is possible to morph, transform and save this graph. It is intended just for illustrative usage as the performance of this part of the software is especially low, so if you try this option on a network with more than twenty units a warning will appear. The network visualization is available at all stages of the training procedure.

The 'Output-Visualization' item is only accessible after a completed sampling and exploitation phase. It offers a visualization of the dynamics in the internal and the output layer during the three phases of the training procedure. If you click the button a new frame, containing two tabs will appear. The first tab is labeled 'Internal states', the second 'Exploitation'. The first tab displays the activity of every unit in the internal layer during the whole procedure in a separate tab. Left-Clicking one of the diagrams switches between local and global scaling, a right-click shows the data table of this special unit. The second tab shows the activity for the output unit(s) during the exploitation phase (as well as the desired output) in separate tabs, the last sub-tab contains a mixed view of all output units. If you perform a left-click at any of the diagrams, you can edit the visualization options, a right-click opens a dialog that allows you two save the graphic as image.

3.1.2 Load existing ESN.

If you click this item a file-chooser dialog will appear. Now you have to choose a valid network-description file, at the right site of the dialog you can see a preview window. Network-description files start with two file paths, followed by the number of time-steps for the washout phase, the sampling phase and the exploitation phase. After selecting a valid file the network is generated and a new frame appears, similar to the one described in the previous section.

3.1.3 View and merge output.

If you do not want to load a whole network, but instead want to view the output of one ESN, or to compare different ESNs, you can use this item. Clicking it will show a file-chooser dialog that allows multiple file selection. All valid files will be load and merged. After this a new frame appears that contains three tabs, one for each phase of the network training procedure. Every tab contains sub-tabs for the layers used in this stage (hence the output layer is only visualized in the exploitation tab). There are sub-tabs for every unit in the selected layer, as well as a tab containing all units in a single figure. The mean activation is shown as well. A left-click on a diagram opens a dialog that let you view the data-tables or edit the visualization, a right-click allows to save the current diagram as an image.

3.1.4 Setup and run scripts.

This item was intended for automation issues, as it is impossible to control the software via setting-files at the moment. This feature will be replaced in the further development of the software, so the following description is rather short. If you click this item a new frame appears. It consists of two parts, a so called ‘editor’ and a console, three buttons are on the bottom of the frame. With these tools it is possible to write and compile a Java class and attach in to the software during run time. To get this to work you have to extend the GeneralScript-class, a method-stub performing this should appear in the editor. The only parameter is a so called SettingsGui-object, providing the possibility to manipulate settings and to iterate (for instance) over a special parameter and generate output. This is not user-friendly as it is necessary to know the API of the software, which involves the examination of the code or at least the documentation. Hence this feature will be replaced by a control via setting-files. If you are interested in examples take a look into the files contained in the scrips directory.

3.2 The Training Options

There are two options available. At first the inspection of existing input-output samples and secondly the possibility to create new training sequences.

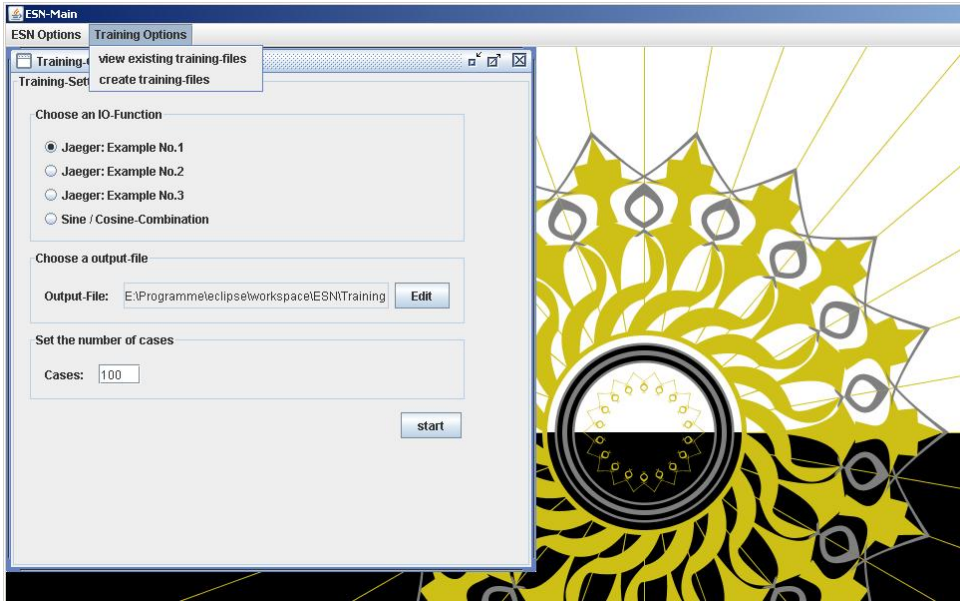

Figure 3: Available items in the Training Option Menu, as well as the frame allowing the creation of training files.

3.2.1 View existing training-files.

Click this item to obtain a visualization of an existing training file. If you choose an invalid file, the appearing frame will inform you that the current selection is erroneous. If the file was valid the frame should contain different tabs, one for the input and one for the output time-series, as well as a tab containing a combined visualization of input and output. The options for the figures are as usual: A left-click let you edit the settings or view the data and a right-click allows you to save the diagrams as images.

3.2.2 Create training files

This item offers the possibility to create new input / output samples to train networks with. Figure 3 shows the frame appearing after clicking this item. The first step is to choose an io-function, by default there are four functions implemented, how to add your own functions is covered in the appendices. Dragging the cursor over the different functions should enable tool-tips that provide some information about the current function. After selecting a function you have to choose a file destination for the data, the

default value points to a file in the local hierarchy. At last you have to choose the number of time-steps, the default-value is 100. Clicking the start button will generate the data at the chosen location. Of course it is possible to create such input-files in different ways, in future versions of the software there should be a formula interpreter to facilitate the generation of io-functions without programming them directly as it is the case at the moment.

4 Examples

The following section includes four examples to illustrate the usage of the software. The necessary parts of the interface will be described repeating and extending some parts of the upper introduction. The first two examples are taken from Jaeger's tutorial about training ESNs (Jaeger, 2002). The last one uses a two-dimensional output layer.

4.1 Example 1: Jaeger's sinewave-generator

In this example an ESN is trained to generate a simple sinewave. The first task is the data-generation. Click on the 'Training-Options' menu in the main-screen and select the 'create-training' files item. Now select the 'Jaeger: Example No.1 button' (a tool-tip should inform you about the output-function). If you want to, you can change the output-file, there should be a default file selected. At last you have to choose the number of sample-steps, 300 should be enough for the current task. One click at the start button will create the input / output samples at the desired file.

If you want to check the generated data-sample select the 'view existing training-files' item from the 'Training-Options' menu and select the file with the generated data. Now you should see a new frame with three tabs: 'Input', 'Output' and 'IO-Combination'. There are sub-tabs in every tab: One for every input or output dimension and a global tab (all dimensions in one figure). If you select the 'IO-Combination' tab, you will see the combined input and output samples: A straight line as input and a sinewave as output (actually input is not necessary in this example, but we will discuss the consequences of using an input layer in a later part of the paper using these data). A right-click at the diagram will open a dialog, allowing to save the diagram as picture if you need the figure for some reason, a left-click enables you to edit the graph (disable the drawing of the means etc.).

Now we can start with the training-procedure. Select the 'create new ESN' item from the 'ESN Options Menu'. This should open a new frame with an interface that allows to change the default settings. As we do not need an input layer in this example, we want to create a network without input units. Therefore go to the 'Network Options' and select the 'Input-Layer' leaf and make sure, that the 'use an input layer' flag is not selected.

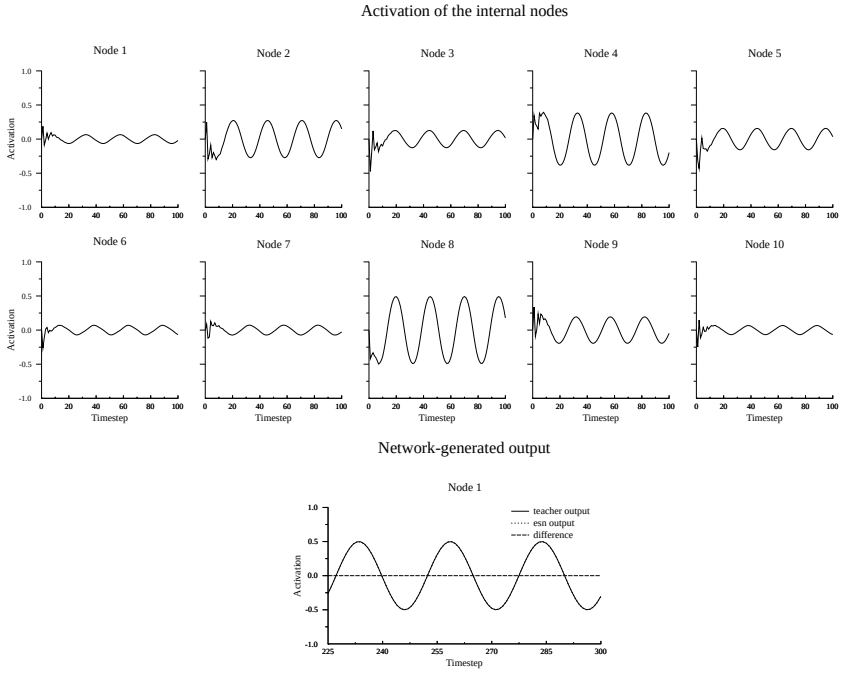

Figure 4: Activation of the ten internal units during the first 100 time-steps of the training procedure and the activation of the output unit during the exploitation phase compared to the desired output.

The only thing we have to change for the internal layer, is the spectral-radius: We have to change the default-value of .8 to .7. That is all we have to do, if you changed the sample file, you have to select it in the 'IO-Settings', there you can also change the default output-directory. Now click at the button labeled 'start'. A dialog should appear, that asks you to distribute the training-data over the initial washout, the training and the exploitation phase. By default the ratio is as follows: 1 to 2 to 1. So in our case of 300 sample-steps, the default-setting leads to 75 steps for the washout, 150 training- and 75 exploitation-steps. If you don't want to change this click the 'Apply' button.

There should be a new tab beside the 'ESN-Settings' tab called 'ESN'. There are four sub-tabs, one for each layer (note that there is no tab for the input layer as we decided no to use one in this example) and one tab for the two errors: One for the sampling and one for the exploitation phase. If you now click the 'run Training' button, washout and sampling will take place. If you select the 'output-layer' sub-tab you can see that there are now estimations for the weights, as well as a value for the sampling error (should be very small). If you click the 'run Exploitation' button, the network tries to generate the desired output for as much time steps as you have chosen for the exploitation-phase. Now the error sub-tab also contains a value for the exploitation error. To obtain a visualization of the output select the 'Output-Visualization' item from the 'Network-Visualization' menu. What you see should be similar to the diagrams in figure 4. As can be seen in the figures displaying the course of activation of the internal units, the initial chaotic flow of activation quickly adapts to the propagated sinewave. Every unit adapts in a unique, but in every case sinewave-like fashion. The bottom panel of figure 4 displays the activation of the output unit during the exploitation-phase: The desired dynamic and the network generated output are nearly identical, the error is around zero.

4.2 Example 2: Jaeger's tunable sinewave-generator

This example is similar to the second example given by Jaeger (Jaeger, 2002). The input is a slow sinewave, the desired output is a sinewave with a fluctuating frequency, depending on the input. Again we have to create the input / output samples at first, follow the steps given in the previous section, but select the 'Jaeger: Example No.2' button and generate 1000 time-steps.

Now select the 'create new ESN' item from the 'ESN Options Menu'. There are different settings we have to change. From the network options select the 'Input Layer' leaf and make sure that the 'use an input layer' flag is checked. As this sample is more complex than the last one we will need a bigger dynamic reservoir, therefore click on the 'Internal Layer' leaf and set the size parameter from ten to hundred. In our runs, we also changed

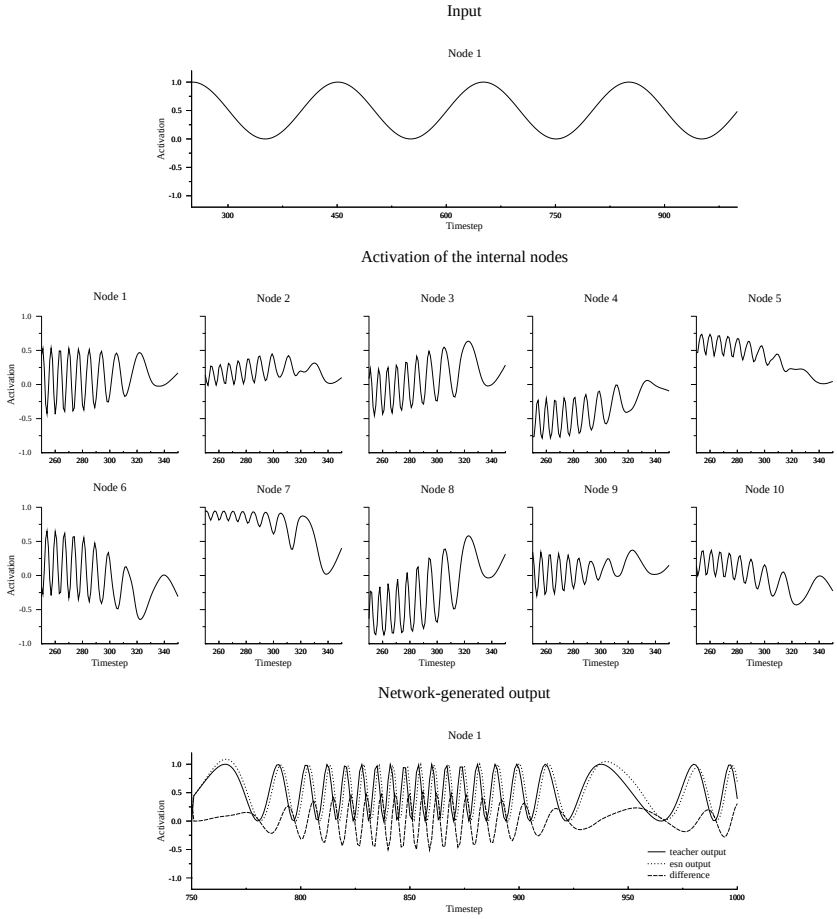

Figure 5: The top panel shows the input during the sampling and the exploitation-phase: A slow sinewave. In the middle panel the flow of activation of the first 10 units during the first 100 time-steps during the sampling-phase is shown. Amplitude and frequency are modulated according to the input. The bottom panel displays the network output compared to the desired output. As can be seen in the graph, the approximation is not optimal, the frequency-shifts in the network output are slightly too slow, but the pattern is still reproduced quite well.

the spectral radius to .77. At last, make sure that the transfer-function parameter in the 'Output Layer' leaf is set to 'ID'. In contrast with Jaeger's example, we use an identical transformation instead of a hyperbolic tangent transformation here. If the hyperbolic tangent was used, the data should be scaled between -0.5 and 0.5 to allow successful learning.

Next, check if the input file is located at the path displayed in the 'basic IO' settings. Due to the complexity of the problem, it is possible that the optimization of the output weights get stuck in local optima. To prevent this, we use noise in this example. From the 'Math' settings select the noise leaf and set the noise-type parameter to 'EQUAL'. A new panel should appear. Now set the 'Lower-Bound' parameter to -.0005 and the 'Upper-Bound' parameter to .0005. This will result in an equal distributed noise term that is added to the activations of the units of the internal layer. Please note that these settings are far from being perfect for this problem. About one of five networks generated with them performs equal or better than the example-network shown in figure 5.

Now click the start button, again a dialog appears that ask you to distribute the time-steps over the three phases. We can use the default-settings, in the case of 1000 time-steps, 250 are used for the washout and the exploitation-phase and 500 are used for sampling. Apply these settings and train the network by clicking the 'run Training' and the 'run Exploitation' button. In most of the cases the mean squared error for the training-phase is quite low in contrast to a more or less high error for the exploitation-phase. As mentioned in the introduction a good approximation is not guaranteed with these settings, if the performance of the network is still too bad after some resets (in this example resetting the network can be useful as a random error is applied during every sampling-phase, hence the resulting output weights are not equal), go back to the 'ESN-Settings' tab and generate a new network.

Figure 5 shows the results obtained with a sample network. The approximation is not perfect and the mean squared error during the exploitation-phase was around .03. The behavior exhibited by the sample network is typical for this example: At the beginning of the exploitation the networks matches the desired output quite good, but during the first frequency-shift the discrepancy becomes bigger with increasing frequency, it seems that the network is incapable to adapt its frequency so quickly. A typical pattern for failed exploitation is an initial phase with a quite good approximation and a breakdown during the second frequency-shift.

4.3 Example 3: Learning of a combined output function

The previous example demonstrated the ability of ESNs to adapt to simple and complex one-dimensional output dynamics. In the current example the task includes the reproduction of a combined sine- and cosine-wave output

function. As in the other examples we have to create the input / output samples at first. The steps are equal to the other examples with the only difference that you have to select the 'Sine / Cosine-Combination' button. 1000 time-steps should be enough.

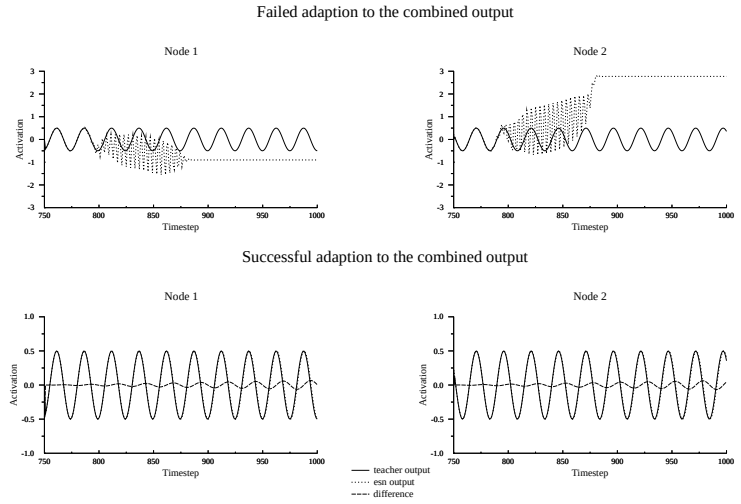

Figure 6: Example output for the two output units from two trained networks. The upper panel shows a failed attempt to reproduce the output. As the two output functions are very similar, failed adaption to one of them increases the chance of failed adaption for the second. The lower panel displays a successful reproduction as it is the typical case for the reported settings. The quality of the fit decreases slightly over time.

Now create a new ESN. The following settings were applied to create the example shown in Figure 6, which generated suitable solutions in about two thirds of the cases. At first, uncheck the 'use an input layer' flag in the input layer leaf of the network settings. Set the number of units in the internal layer leaf to 25, the default setting for the spectral radius can remain unchanged. At last set the number of output units to two and make sure that the 'Interconnect the output units' checkbox in the 'output-layer' leaf is not selected. As the desired output functions can be easily transformed into each other, interconnections between the output units would make the internal units obsolete as the output can be combined with a single teacher forced input (you can test this by selecting the checkbox and setting the internal layer unit count to one, the errors should be extreme low as well as the estimated weights for the connections between the one internal and the two output units). Now hit the start button, the default distribution of the time steps can remain unchanged.

Figure 6 shows the results of two example networks. The erroneous behavior shown in the upper panel of Figure 6 is one example for failed

adaptions in this task, wrong amplitudes and frequencies are typical as well.

5 Network parameters

The following section provides an overview of some crucial parameters for training ESNs. For a detailed discussion see, for example, Jaeger and Haas (2004); Jaeger, Lukosevicius, Popovici, and Siewert (2007).

5.1 Spectral-radius

The main characteristic of an ESN is the so called 'echo state property', a mathematical definition can be found in (Jaeger, 2002), or (Jaeger et al., 2007). An intuitive definition holds (see (Jaeger, 2002) as well) that the current network state has to be uniquely determined by the history of the input and the (teacher-forced) output (given that the network has been run for a long enough time). The echo state property depends on some prerequisites of the dynamic reservoir, which are determined by the connectivity of the internal layer. Even if there is (at the moment) no known necessary and sufficient algebraic condition that allows to decide whether a given network has the echo state property or not, it is possible to formulate a sufficient condition for the non-existence of echo states. In short, it holds that the network has no echo states if the largest absolute value of the Eigenvector is larger than one. In this case, the randomly selected strengths of the connections are high enough to produce growing oscillations. The internal states of a network without the echo state property will be influenced by the initial network states (no wash out will take place) and hence will not resemble the pure input / output sequence. A simple test of this property includes a single impulse-like activation of the internal units: If the network possesses the echo state property, the resulting activation should diminish until it is zero.

Figure 7 shows the dynamics of the internal states and the resulting output for two networks with different spectral radii. There are two noticeable features of the dynamics of the internal states in the case of a spectral radius of 2.5: At first the high frequency of the resulting oscillation and secondly the high amplitude. The oscillation seems to be quite chaotic as well, even if it is somehow sinewave shaped. The left graph in the bottom panel of Figure 7 displays the output computed with these states. Of course there are strong deviations from the intended (teacher) output.

There might be a broader or smaller range of suitable values for the spectral radius depending on the current task. In practice the value of the spectral radius determines the timescale of the dynamic reservoir. As a rule of thumb the spectral radius should be small for fast, short-memory tasks and larger for slower, long-memory tasks (Jaeger, 2002). The term 'memory' in this case refers to the influence of previous sequences on the

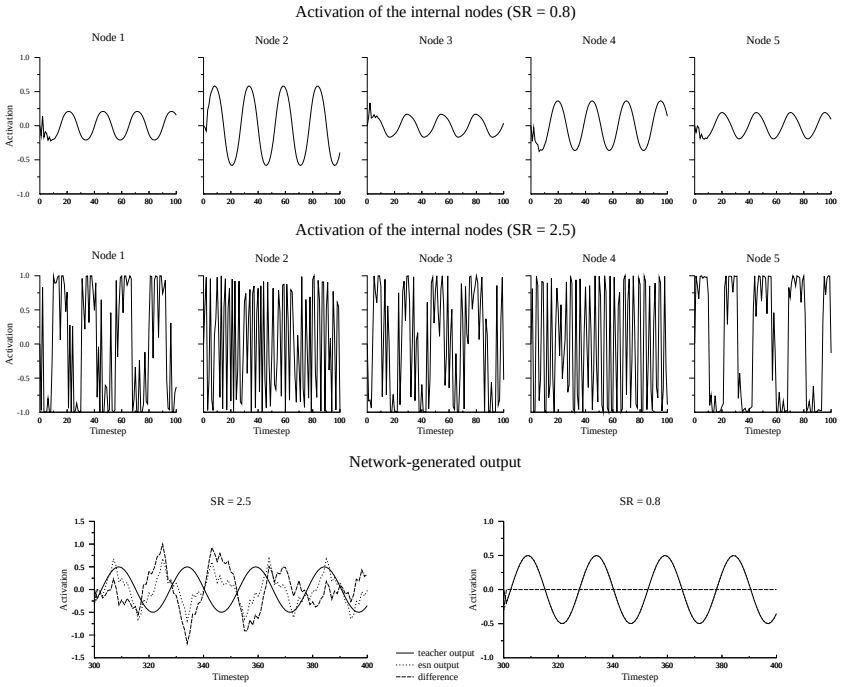

Figure 7: The first five internal states and the network generated output for two networks with different values for the spectral radius. The settings are the same, except the spectral radius. As can be seen in the middle panel, a very high spectral radius leads two unstable activation-flows in the internal states. The probability to reproduce the correct output is very low given these dynamics (bottom panel left graph).

current network state, that is, how long former activation has an influence on the current dynamics. Another remarkable feature of the spectral radius is the non-linear influence on the timescale. Empirical observations imply an exponential dependency of the timescale on $1 - \alpha$, where α denotes the spectral radius (Jaeger, 2002).

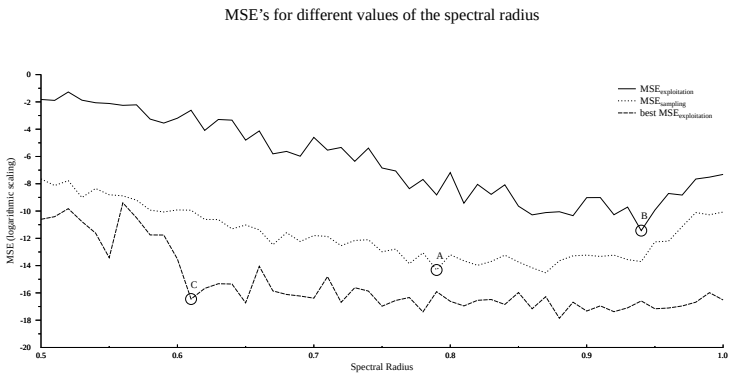

Figure 8: Performance of networks with different spectral radii for the same task. Every data point represents a mean-value of the log_{10} MSEs of 50 networks. Additionally, the best $\text{MSE}_{exploitation}$ found in the 50 networks is displayed. Please note that the figure tends to overestimate the quality of the network-solutions: As the original MSEs were logarithmized, the influence of bad solutions is rather low.

It is possible to optimize the spectral radius for a given task. Even if the range of suitable values is rather broad: If an ESN with a spectral radius of 0.8 performs well, it can be expected to perform equally well with a spectral radius of 0.7 or 0.85. Figure 8 shows an example for the influence of the spectral radius on performance. The task was the sinewave-generator from example 1. All networks had the same settings: 20 units in the internal layer, no input layer, a *tanh* transformation in the internal layer and an identical transformation in the output layer. 50 networks were trained for the different spectral radii and the mean of logarithmized $\text{MSE}_{exploitation}$ and $\text{MSE}_{sampling}$ was calculated. Furthermore the best logarithmized $\text{MSE}_{exploitation}$ values from the 50 networks were stored. The marked points 'A' and 'B' indicate the smallest values found for the $\text{MSE}_{sampling}$, the $\text{MSE}_{exploitation}$, respectively. As can be seen from Figure 8, these two points do not coincide, even if the performance of the networks is nearly equal for spectral radii between 0.78 and 0.97. Point 'C' marks the best threshold for the spectral radius in the given task for the best $\text{MSE}_{exploitation}$ value achieved .

In sum, smaller spectral radii should be applied if faster dynamics are desired, but larger values (closer to one) for slower dynamics. However, the exact choice of the spectral radius appears not to be of crucial importance. Even if it might be useful to optimize the spectral radius for a given task,

in most cases the range of acceptable values is rather broad.

5.2 Usage of an input layer

Figure 1 shows that the output layer is connected to the input layer by default if an input layer exists. Hence the flow of activation in the input layer influences both the internal, as well as the output layer. If this activation is unrelated to the current task, the incorporation of these activations may decrease the performance of the network. You can examine this by using an input layer for the sinewave-generator task from example 1. By default the input / output sequence generated by the software includes a linear function for the input. If this activation is mapped into the network it will superimpose the intended sinewave, which can be seen in the flow of activation in the internal states as well as in the behavior of the network during the exploitation phase. On the other hand, if the input is equal to the teacher-output (simple input-output matching) the estimated weights for the output layer will be very low for the connections to the internal units and high for the connections to the input units, making the dynamic reservoir obsolete. Hence the usage of an input layer depends on the given task. Jaeger (Jaeger, 2002) proposes the usage of a constant input (bias) in cases where the desired output has a mean value different from zero. Sometimes it might be a good idea to let the dynamic reservoir uninfluenced by this bias and map it only to the output-layer, as the bias can force the units in the internal layer to the extreme values of their respective sigmoids.

5.3 Amount of units in the internal layer

As the units of the internal layer are interconnected, there is a continuous interaction between all internal units. This may lead to superpositions in the activations that can not completely controlled by the estimated weights for the output layer. Especially for easy tasks there might be no need for a lot of variance in the internal states. Thus, sparse connectivity might be very useful to provide a large amount of different types of internal dynamics (Jaeger & Haas, 2004).

Given full connectivity, Figure 9 shows how the performance in the sinewave-generator task depends on the amount of units in the internal layer. The settings for all networks were the same except for the size of the dynamic reservoir. As can be seen from Figure 9, the performance is very good for internal layers with 10 to 20 units and decreases for larger dynamic reservoirs.

Another problem associated with large dynamic reservoirs is overfitting: If the ESN is trained with noisy data, a large dynamic reservoir may lead to the adaption to random fluctuations resulting in poor generalization abilities of the trained network. The optimal size of the internal layer depends on the

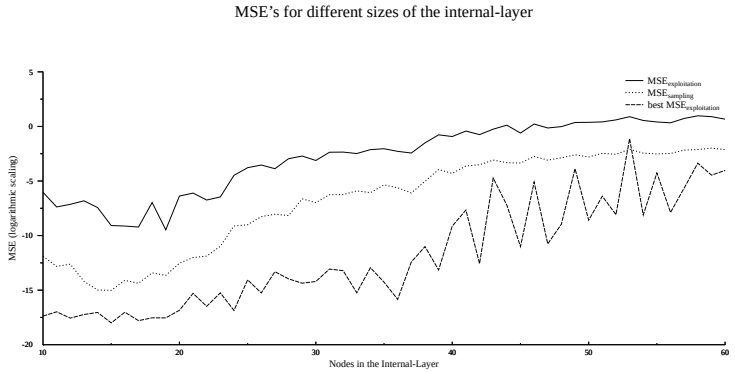

Figure 9: The influence of the size of the internal layer on the performance in the sinewave-generator task. The curves display the means of the log_{10} MSEs of 50 networks for the different sizes (the amount of units in the internal layer) plotted on the x-axis. For every size the lowest $\text{MSE}_{exploitation}$ found in the networks is displayed as well.

task. It is a parameter that might be optimized, but it seems also possible to increase the size until the performance on test data decreases (Jaeger, 2002).

5.4 Self-Recurrence in the output layer

For higher dimensional output one has to decide if the units in the output layer should be interconnected. This might be a good idea if the activation of one unit has a predictive value for the other units, but this can lead to interferences as well. On the other hand, if the activations of the units can be completely predicted from each other, the internal layer is no longer necessary, resulting in very low estimates for the corresponding weights and high values for the weights interconnecting the units in the output layer with each other and themselves. You can try this by running the third example with interconnections in the output layer and a size of one for the dynamic reservoir. The performance during the exploitation phase will be very good, even if the output is not combined using the activation of the remaining unit in the dynamic reservoir, but simply by the connections between the two units in the output layer. Once again it depends on the task if the usage of interconnections in the output layer are useful or not.

6 Discussion

The ESN architecture offers an elegant approach to the difficulties involved in time-series prediction tasks because it avoids most of the common problems of training recurrent neural networks. The described Java implemen-

tation is able to reproduce results obtained with MATLAB or C++ and it is hoped that it is useful for future research studies on ESN as well as for illustrative purposes.

In the future, we intend to develop the software further in order to increase its flexibility and usability. Another important point is the missing possibility of online-learning in the current form of the software. This will be improved in further releases, so the user can choose between batch- and online learning.

We used two third-party packages for the current implementation. The linear algebra is performed with the JAMA package, developed in cooperation by the National Institute of Standards and Technology (NIST) and The MathWorks. As the package was released to the public domain there exist no licenses for this package. The graph-like visualization of the networks was obtained using the JUNG package, protected by the Berkeley Software Distribution (BSD) license.

The code is distributed for academic purposes with absolutely no warranty of any kind, either expressed or implied. We are not responsible for any damage from its proper or improper use. If you are interested to work with the source code, the appendices offer an overview of the structure and the interfaces necessary to include customized functionalities.

Even if the software will be refactored and some aspects will not be included in further releases, feel free to use, modify and distribute the code with an appropriate acknowledgment of the source. In all potentially resulting publications please include the following citation:
J. Lohmann & M.V. Butz (2009), Echo-State Network in Java: Introduction, manual, and evaluations (COBOSLAB Report Y2010N002). Retrieved from University of Würzburg, Cognitive Bodyspaces: Learning and Behavior website: http://www.coboslab.psychologie.uni-wuerzburg.de/.
Also, please report any bugs or other inconsistencies in the source code to one of the authors.

References

Cruse, H. (2006). *Neural networks as cybernetic systems (2nd and revised edition).* Brains, Minds and Media, Bielefeld, Germany.

Jaeger, H. (2001). *The "echo state" approach to analysing and training recurrent neural networks* (Tech. Rep. No. GMD Report 148). GMD - German National Research Institute for Computer Science.

Jaeger, H. (2002). *Tutorial on training recurrent neural networks, covering BPTT, RTRL, EKF and the echo state network approach* (Tech. Rep. No. GMD Report 159). Fraunhofer Institute for Autonomous Intelligent Systems (AIS).

Jaeger, H., & Haas, H. (2004). Harnessing nonlinearity: Predicting chaotic

systems and saving energy in wireless communication. *Science*, *304*, 78 - 80.

Jaeger, H., Lukosevicius, M., Popovici, D., & Siewert, U. (2007). Optimization and applications of echo state networks with leaky integrator neurons. *Neural Networks*, *20*(3), 335-352.

A Package Structure

The following section provides an overview of the package-structure, as well as of the dependencies between the packages constituting the software. Since the structure partially developed over time, there are some slightly odd dependencies and violations of the general MVC-model. The following description will help to work with the software, nonetheless. All UML diagrams were created using UModel from Altova, members and attributes are not displayed to keep the diagrams as clear as possible.

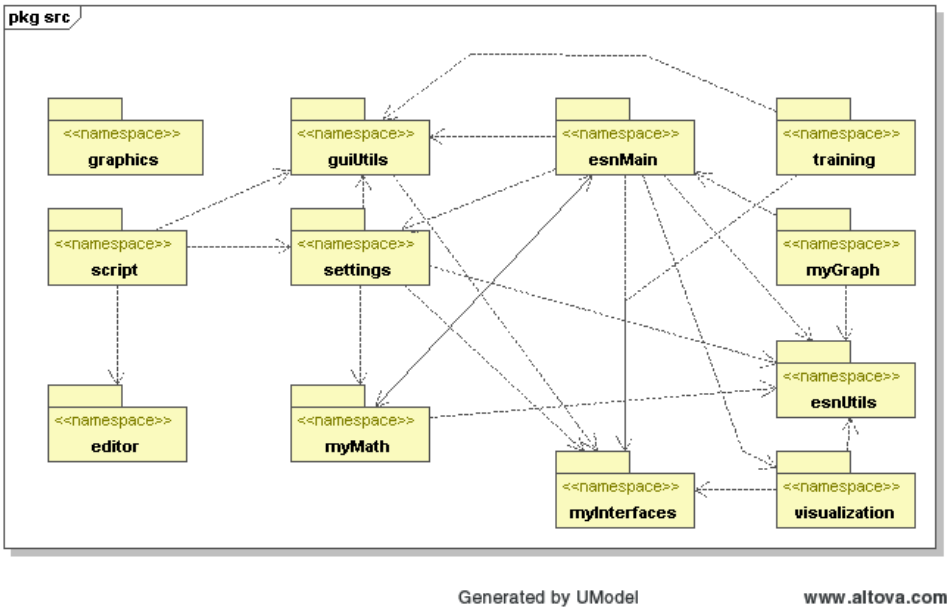

Figure 10: The package structure and the dependencies between the packages of the software. There exist circularities, implying a sub-optimal design. For instance the `esnMain` package depends on the `visualization` and the `myInterfaces` package, while the `visualization` depends on the `myInterfaces` package as well.

There are three distinct parts that constitute the whole program: The main part that realizes the setup, the training and the exploitation of the ESNs, as well as the visualization. The second part creates, saves and visualizes the input / output sequences used for training the networks. The last part is the editor that is used for scripting issues in the current form of the software. As can be seen from Figure 10, these three parts are not

as distinct as they maybe should, but the following description separates them as much as possible. Note that nearly all other packages depend on the `guiUtils` and the `myInterfaces` packages.

A.1 The main Program

The so called `esnMain` package contains the basic GUI elements as well as the class that provides the ESN functionality. Figure 11 shows the different classes in this package. The `EsnNetwork` class performs the setup and the training of ESNs, a view to this functions is provided by the `ESNSubGui` class. The `MainGui` class hosts the whole GUI and provides an `JFrame` to store all the `InternalFrame`s used for the different functions accessible via the menu of the main frame (see the different action-methods in the source code of the `MainGui` class). The `ESNOutputViewer` allows to inspect and to merge different existing output-files via tabbed-panes. The data stored in the selected input-files are read and merged using the `OutputMerger` class. The `ESNLoader` class is used to load and visualize existing ESNs.

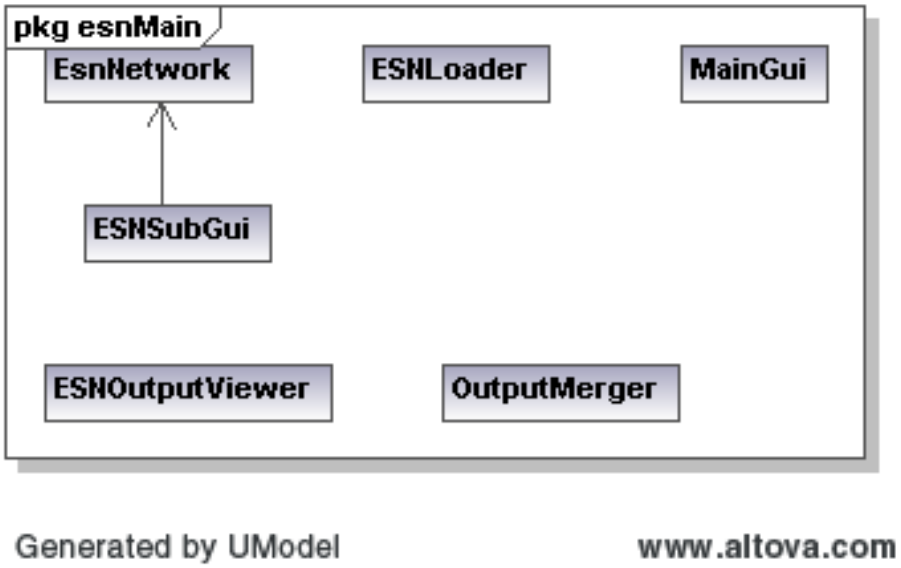

Figure 11: The classes of the `esnMain` package: The model is realized in the `EsnNetwork` class, the other classes provide the basic functionality of the GUI (a mix of model, view and controller).

The `esnMain` package depends on the `myMath` package and vice versa (see Figure 10). This circularity is caused by the fact that nearly all methods of the classes in the `myMath` package work on instances of `ESNNetwork` and on the other hand, all the mathematics that are necessary for the training of ESNs are stored in this package (see Figure 12). The `NetworkFunctions`

class contains only static methods used for training the ESN, to calculate the MSEs and to apply noise. What kind of noise is used in the update of the internal states depends on the settings of a given ESN, the necessary calculations are performed by the `Noise` class. This class also stores an enumeration (`NoiseTypes`) describing the type of noise, instances of this enumeration are used in the settings of the ESNs.

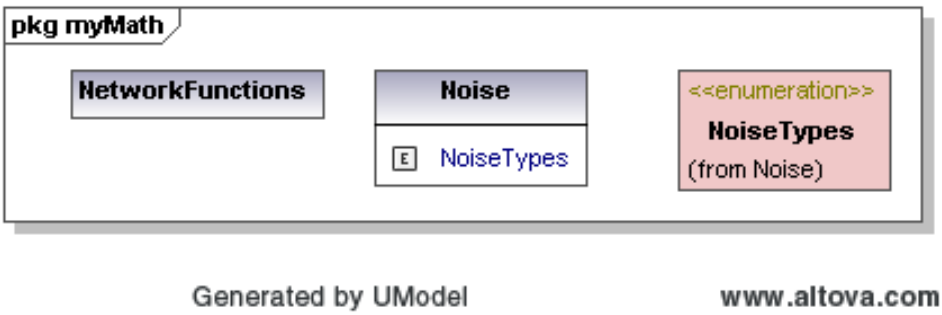

Figure 12: The classes of the `myMath` package. The `NetworkFunctions` class provides the functions involving linear algebra necessary to train the ESNs. Hence this class depends on the `JAMA` package. The `Noise` class allows the application of noise on given states. A enumeration denoting the different types of noise is nested in the `Noise` class, these types are properties of an ESN.

The `esnMain` package as well as the `myMath` package depend on the `esnUtils` package (see Figure 10). This package stores classes that hold representations of the different layers of the network. There exists an abstract `Layer` class (see Figure 13) that is the parent of the different layers. All classes extending the `Layer` class have to implement the `initializeWeights()` method. The classes representing the input and back-projection layer are nearly equal. The `InternalLayer` class has some special features: a value that indicates the connectivity between the units, a possible double array representing an explicit representation of the probability distribution for the connectivity, and a value for the spectral radius. The `OutputLayer` class differs from the other implementations of the `Layer` class by the existence of a flag indicating if the units should be self-recurrent (if false all recurrent connections in the output layer are excluded). All layers have a property called `func` from the type `TransferFunctions`. These are only meaningful for the `InternalLayer` and the `OutputLayer` as during the update of the states of these two layers a transfer function is applied. The `TransferFunctions` class holds the methods used for applying the functions as well as an enumeration (`functions`) indicating the existing functions. Instances of this enumeration are utilized to describe the properties of the layers in the settings of a given ESN. The last class in this package (`InternalState`) is used to store information about the network generated through the training process. The attributes of this class indicate the stored information: The

current input as well as the intended / teacher output and the activations of the units in the internal layer are stored in double arrays, an error value is reserved for the network generated output (the name is confusing as no error-value in terms of a difference between teacher and network output is stored) and a String contains the information from which part of the training process the data was obtained (wash out, sampling or exploitation). There exists a `toString()` method to retrieve a printable representation of the data that is utilized to store it in files (see the implementation of the `printOutput()` method in the `ESNNetwork` class).

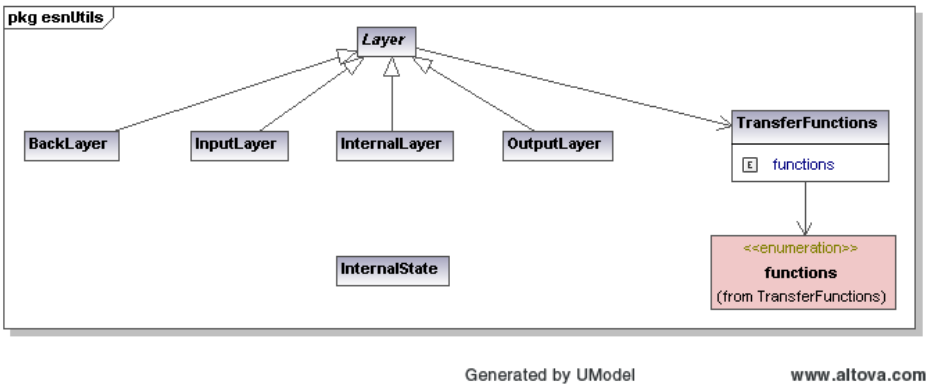

Figure 13: A class diagram for the `esnUtils` package. Most classes extend the abstract `Layer` class. The `TransferFunctions` class provides the methods used for the application of transfer functions in the update of activations in the internal and the output layer. The `InternalState` class allows the storage of information obtained during the training process.

The `settings` package (shown in Figure 14) contains all the properties of an ESN incorporated in widgets. The `SettingsGui` class extends the `InternalFrame` class providing the frame for the visualization of the different properties in a tree-like manner. The `MyNode` class is used to store the data displayed in the leafs of the settings-tree. The other classes of the package (`IOSettings`, `MathSettings` and `NetworkSettings`) store the information of the different types of settings. They are also able to create a view to these settings using `JPanel` instances. The editable variables are stored in enumerations as well as in the attributes of the classes. This redundancy is used to read settings from files (so the files do not have to match a specific structure, but they have to contain the appropriate keywords). As can be seen in Figure 14 the three classes implement an interface called `GeneralSetAndGet`. This interface is used to set and get attributes from this classes via string-based identifiers, these identifiers coincide with the names of the enumeration instances (see the implementations of the `fromStringArray()` methods in the classes and the `GeneralVariable` class in the `guiUtils` package). As the enumerations indicate values for noise

types and transfer functions are stored in other packages, the settings package depends on the `myMath` and the `esnUtils` package (see Figure 10). Due to the fact that the different `Settings` classes are managed by the `SettingsGui` class, it is impossible to access the settings without an instance of the `SettingsGui` class. Hence one has to create such an object every time a ESN is to be trained, even if a GUI is not needed or wanted. The constructor of the `SettingsGui` class might be parametrized with a Boolean that indicates if the GUI should be generated, but this remains a workaround and will be improved in further releases.

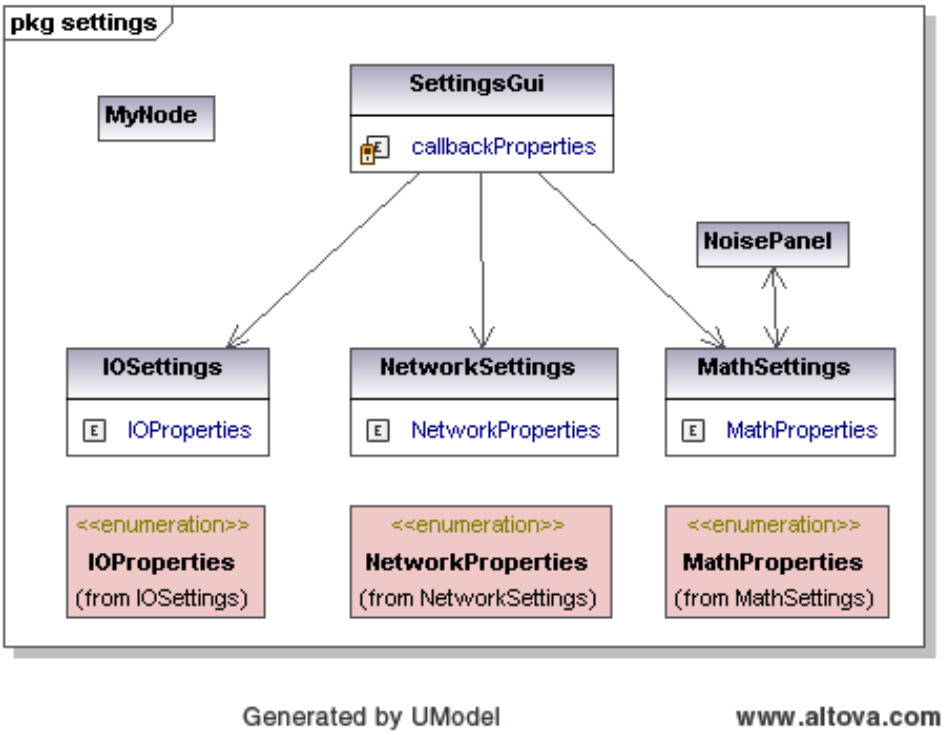

Figure 14: Classes and enumerations stored in the `settings` package.

Another package the `esnMain` package depends on is the `visualization` package. The classes contained in this package provide the possibility to display different aspects of the networks as well as some options that can be performed on the graphs (for instance saving the graphs as pictures or to edit the visualization). As the functionality of these classes is quite constrained and a lot of abstraction can be done, this package will be replaced by a standard graph package (probably JFreeChart). Therefore the package will not be described in detail here (for an overview of the classes see Figure 15).

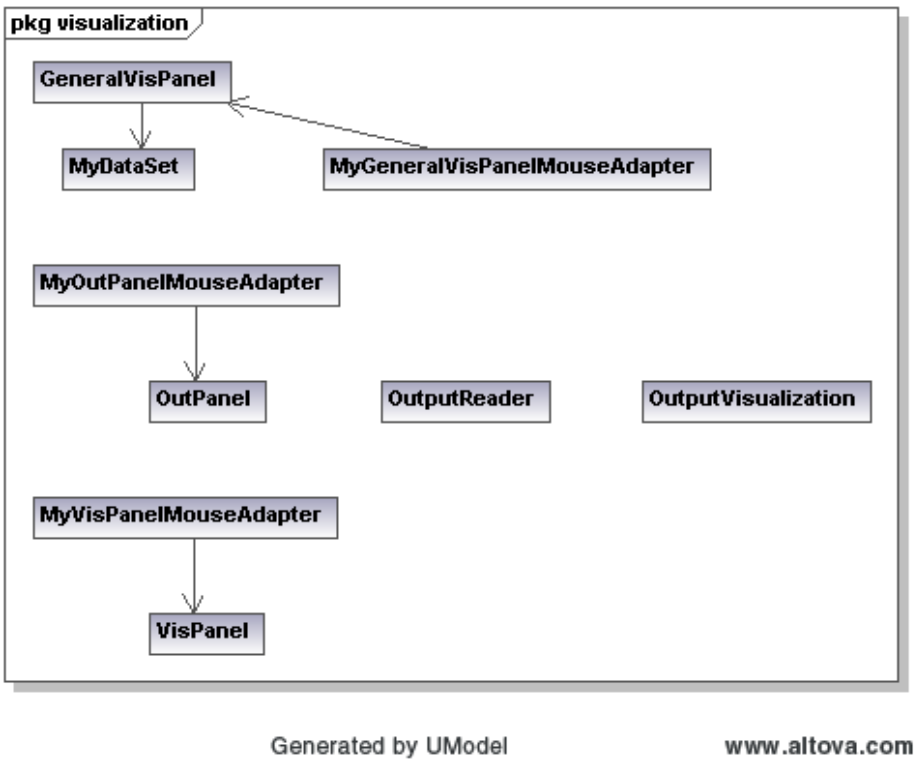

Figure 15: The classes contained in the `visualization` package.

A.2 The creation of input / output sequences

The whole functionality behind generation and visualization of input / output sequences is hold by one package called `training`. This package depends only on the utility packages `guiUtils` and `myInterfaces` (see Figure 10). Three separate tasks are performed with the classes of this package: The generation of input / output sequences and their storage in files, the visualization of existing input / output sequences and a widget making these functions accessible via the GUI. The first task involves the definition of input / output sequences, their calculation and the storage into the file-system. The abstract class `GeneralTraining` defines the methods necessary to generate training sequences (the `generateTrainingSequence()` method) and to use them via the GUI (the `getName()` and `getDescription()` methods). Two attributes are necessary for all implementations of the `GeneralScript` class: A path indicating the file-location where the data should be saved and a number of cases that should be created. The intended function is realized by the heirs of the `GeneralScript` class via their implementation of the `generateTrainingSequence()` method. This method is used by the `TrainingGenerator` to create the data and store it in the file denoted by the `trainingFile` attribute of the current `GeneralTraining` instance. The `TrainingVisualization` class can be used to produce a visualization of existing training files. The `TrainingGui` class displays this visualization in separate tabs using an `InternalFrame`. To generate training sequences via the GUI, the `TrainingGenerator` class allows to choose a function (the known implementations of `GeneralScript` in the package), a file-path, and a number of cases. Figure 16 offers an overview of the classes in the package.

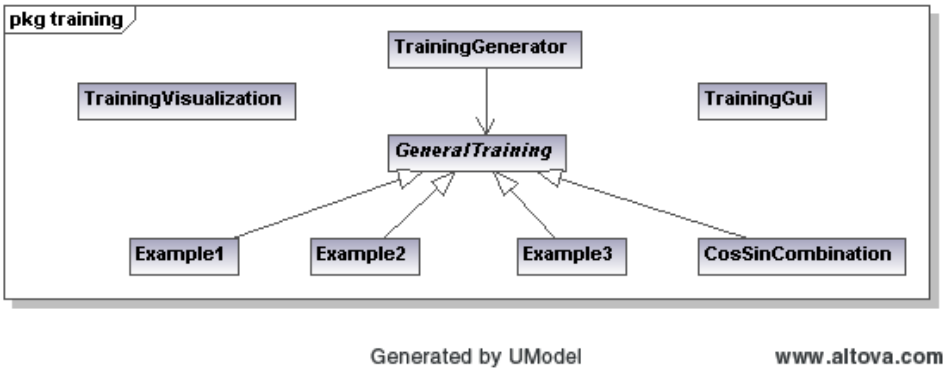

Figure 16: The internal structure of the `training` package. The abstract `GeneralTraining` class is the parent-class for all classes used to generate training sequences. The `TrainingGenerator` class produces the data and stores it, it is also able to create a widget allowing the generation of training files via the GUI. Existing training files can be displayed using the `TrainingVisualization` class in combination with the `TrainingGui` class.

A.3 The Editor

The main functionality of the editor is provided by two packages: The `editor` and the `script` package. The classes in the editor package form a stand-alone application. This application is intended as a small Java-editor to setup and execute 'scripts' (in fact Java-classes that are compiled and executed during run-time). The `editor` package contains all the relevant parts of the editor: Classes that allow the identification and the coloring of the tokens constituting a Java class, as well as a widget containing the colored text (please note that no documentation of the classes store in the `editor` package is provided). The compiler and a console that is used to display compilation errors are parts of the `script` package. The `script` package connects the editor application to the rest of the GUI (see the dependencies of the `script` package in Figure 10). As these packages will not be included in further releases, but will be replaced with packages allowing the control of the software via setting-files, we will not discuss them in detail here. If you want to use the scripting-options, the only relevant class is `GeneralScript`, an abstract class that should be the parent of all costume scripts. It is important to extend this class, as the compiler tries to identify the type of the current script using reflection and the only accepted type is `GeneralScript`.

A.4 Utility packages

As can be seen from Figure 10 most of the described packages depends on two packages, called `guiUtils` and `myInterfaces`.

The `myInterfaces` package contains just three interfaces (see Figure 17). The first one, called `GeneralSetAndGet` allows to set and get attribute values without knowledge about the explicit getter or setter. This is mainly utilized to obtain data from string input. Another case where it is used can be found in the implementation of the `GeneralVariable` class in the `guiUtils` package. This class is parametrized with an instance of `GeneralSetAndGet` and an enumeration value indicating what parameter should be controlled by this specific instance of `GeneralVariable`. Therefore the `GeneralVariable` class can be used to produce Widgets for any attributes of an instance of `GeneralSetAndGet`. The second interface called `NetworkGui` is used by only two classes: `ESNSubGui` in the `esnMain` package that implement it and the `InputValidation` class in the `guiUtils` package that requires an instance of it. This is a relict from the early stages of the development of the software (a quite useless one: A direct parametrization of the `InputValidation` with an instance of a `ESNSubGui` would be easier). It is used to permit the `InputValidation` object to create a new `ESNNetwork` object and give it back to the instance of `NetworkGui` as well as to set the index of the tab containing it accordingly. This interface will not be part of further releases.

The last interface is called `Callback`. The main purpose is the handling of threads. Most threads used in the software require a instance of `Callback` to pass generated data back to the object that needs them. What action should be taken with the data is indicated by the enumeration object that is the first parameter of the `callback()` method. Multithreading in the current form of the software is mainly realized using this interface. Further releases will rely on the ThreadPoolExecutor that is part of the standard Java API, so the `Callback` interface will become obsolete.

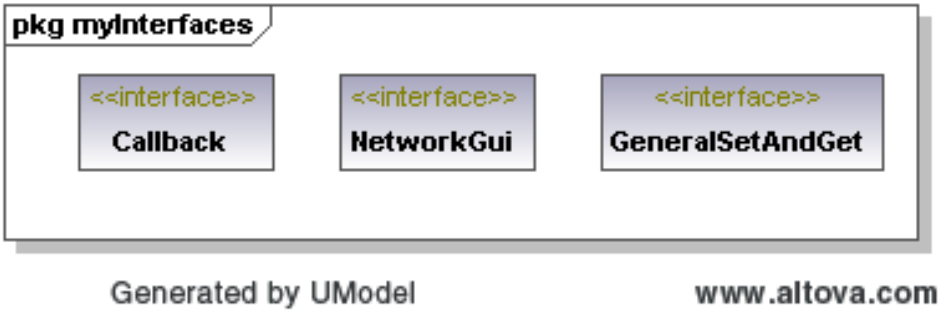

Figure 17: The interfaces contained in the `myInterfaces` package.

The `guiUtils` package contains a lot of classes that are used for tasks related to the graphical user interface. As the functionality is rather heterogeneous it will not be described in detail. An overview of the package structure is given in Figure 18. All classes that have a `Filter` in their name as well as the `LinkedDoubleInputListener` class extend the `DocumentFilter` class from the `javax.swing.text` package. They are all used to prevent invalid input in the text fields used in the GUI (double values if integers are required, string data for values that could only be numeric etc.). Classes with names containing `Table` are used to customize the behavior of `JTables`, like data models or the way the tables are drawn. The rest of the classes are used for different purposes. The `DefaultFileChooserPreview` class generates the accessory component used in some of the `JFileChooser` dialogs that allows a preview on the file content. `GeneralReader` and `GeneralWriter` are two `Runnables` that can be used to read from or write to files. They both require an instance of `Callback` to send the read data back or to notify the object that invoked them. The `GeneralVariable` is used to create default widgets like check-boxes or text fields for given variables. A instance of `GeneralVariable` requires a `GeneralSetAndGet` that is updated if data is changed, as well as a enumeration object that allows the usage of the `generalGet()` and `generalSet` methods. As noted earlier the `InputValidation` class is used to create new ESNs. Finally the `MyProgDialog` class provides the default dialog that is shown sometimes while a thread is working.

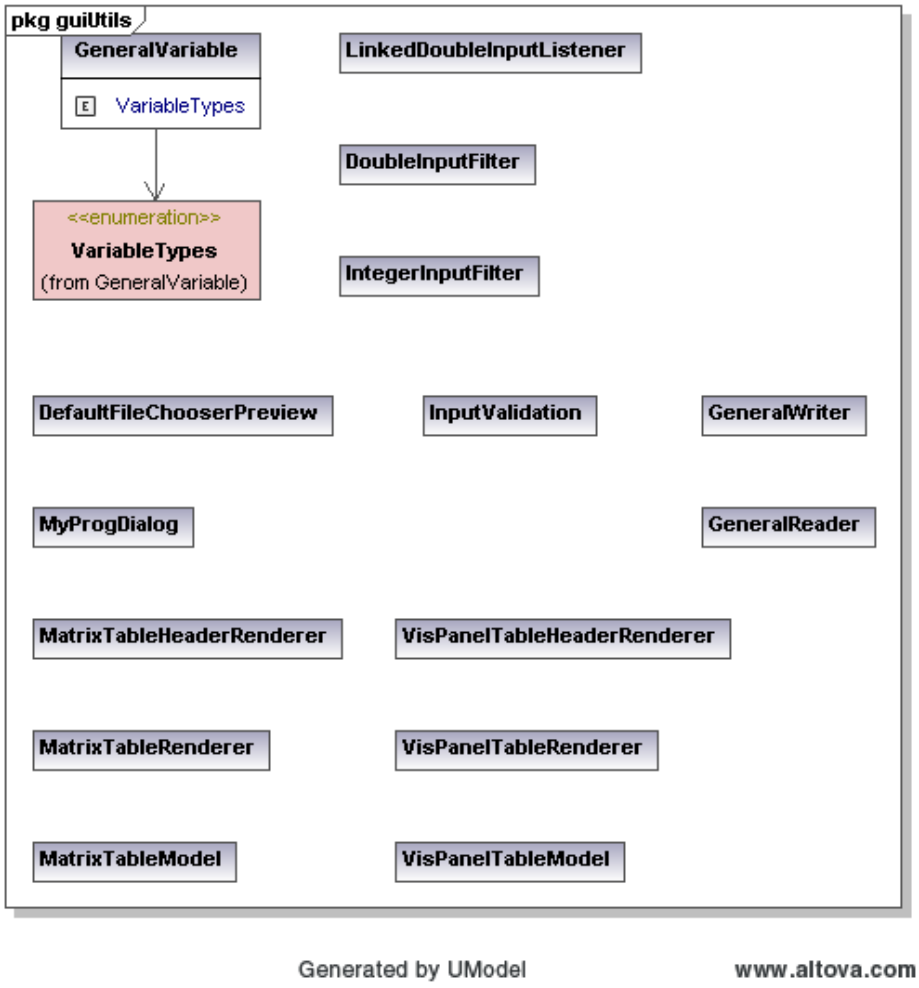

Figure 18: The different classes in the `guiUtils` package.

A.5 Miscellaneous packages

As noted in the discussion we used the `JUNG` package to create a graph-like representation of the different layers constituting an ESN. Figure 19 offers an overview of our implementation. The performance of this feature is quite low especially for larger networks. Hence it is not clear if it will be part of further releases and will not be discussed in more detail.

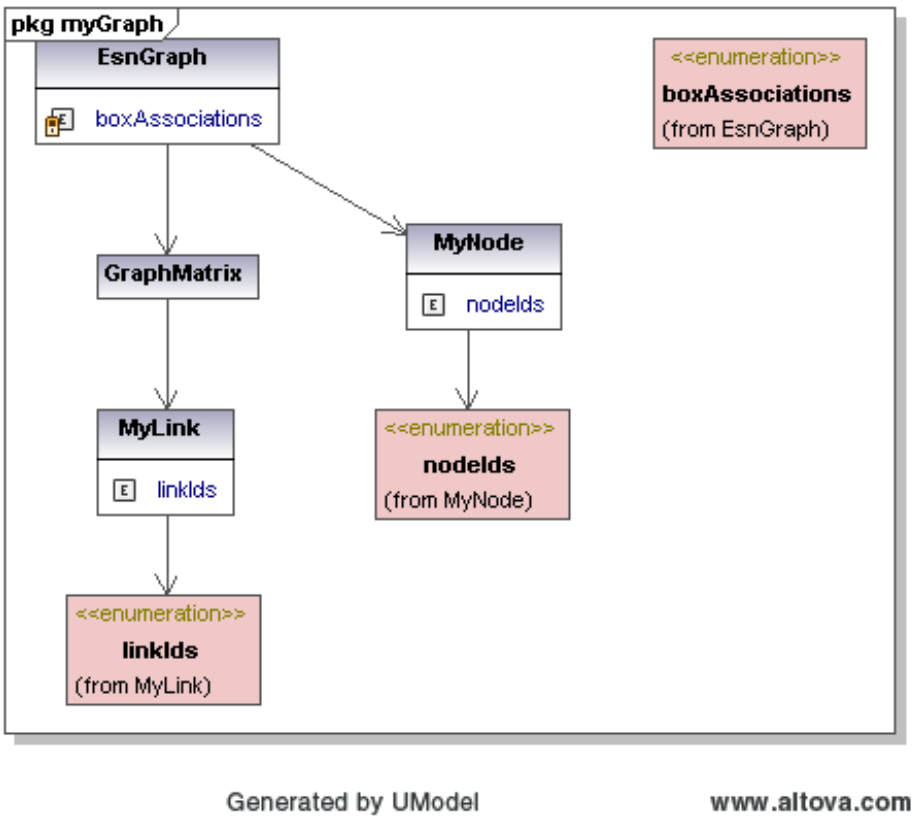

Figure 19: The usage of the `JUNG` package for the graph-like visualization of the ESNs.

The last remaining package is the `graphics` package. It contains one single class, called `WaitPanel`. This class extends `JPanel` and is used to create and display the background image of the main frame. Optionally it is possible to invoke a thread rotating the sun-like picture.

B Extending the Software

Although the whole structure described in the previous appendix is still somewhat bulky, there are some aspects that are candidates for customization: For instance the generation of new training sequences or the imple-

mentation of other noise types than the current ones. The effort to achieve this is quite large in the current implementation. The following example might illustrate this for the creation of a new training sequence. Unfortunately you have to refer to the source code if you want to customize any other features as well.

Figure 16 shows that two tasks have to be solved to implement a new training sequence: At first a implementation of `GeneralScript` have to be written. The second step is to make this new function available in the GUI. The class `GeneralScript` has three methods that should be implemented:

1. `protected abstract double[][][] generateTrainingSequence()`. This method has to be implemented as it is abstract. It should return an array of double values, which contain the training data. The first dimension (that is, the number of rows) represents the number of cases that should be created. Each case consists of values for the input and the intended teacher output (you may take a look into the existing implementations for examples).

2. `public static String getDescription()`. This method should return a String describing the input / output function created by the instance of `GeneralScript`. This method is optional as it is only used for the tool-tip of the button corresponding to this training sequence.

3. `public static String getName()`. This method is optional as well it should deliver a name for the input / output function that is used for labeling the button corresponding to this training sequence.

The existing implementations of the `GeneralScript` class can serve as examples for possible implementations. To make the new function available via the GUI, the `TrainingGenerator` class have to be edited. This is due to the fact that at the current stage of development the existence of possible instances of `GeneralScript` is not checked using reflection. At first one has to create a button for the new type of training. This is possible by editing the `generateButtonPanel()` method in the `TrainingGenerator` class. Please follow the notes given in the source code to achieve this. Secondly you have to register this new button to the implementation of the `ActionListener` interface in the `TrainingGenerator` class. Go to the `actionPerformed(ActionEvent e)` method and follow again the instructions included in the source code. This is a quite cumbersome solution, but further releases will offer a more comfortable way to include costume training sequences.