

MemBrain Help

Table of contents

| | |
|---|----|
| Welcome to MemBrain | 7 |
| Remarks to the Help File | 12 |
| Short Beginner's Tutorial | 12 |
| Adjust MemBrain's settings | 13 |
| Place the neurons for the net | 16 |
| Specify Input and Output Neurons | 17 |
| Connect the neurons | 19 |
| Randomize the Net | 22 |
| Rename the Neurons | 24 |
| Enter I/O Data | 24 |
| Configure a Teacher | 26 |
| Teach the Net | 28 |
| Check Patterns | 30 |
| Examine the Net | 31 |
| Included Examples | 32 |
| Neurons in MemBrain | 33 |
| Adding Single Neurons | 34 |
| Adding Neuron Matrices | 34 |
| Adding Neuron Layers | 35 |
| Adding Decay Neurons | 37 |
| Adding Delay Neurons | 43 |
| Adding Integrator Neurons | 49 |
| Add Differential Neuron | 54 |
| Add LSTM Blocks | 57 |
| Moving Neurons | 60 |
| Aligning Neurons or Groups | 61 |
| Changing Neuron Properties | 63 |
| Quick Activation | 68 |
| Neuron Auto Naming | 68 |
| Activation Functions | 69 |
| Neuron Model And Operation | 71 |
| Links in MemBrain | 74 |
| Adding Links Between Neurons | 77 |
| Single Connections | 78 |
| Connect FROM Extra Selection (FULL) | 79 |
| Connect FROM Extra Selection (RANDOM) | 80 |
| Connect FROM Extra Selection (1:1) | 82 |
| Connect FROM Extra Selection (Matrix based) | 84 |
| Connect TO Extra Selection (FULL) | 88 |
| Connect TO Extra Selection (RANDOM) | 89 |
| Connect TO Extra Selection (1:1) | 89 |
| Connect TO Extra Selection (Matrix based) | 89 |
| Add Outputs to Selection | 89 |
| Add Inputs to Selection | 90 |
| Interconnect all neuron layers downwards | 90 |
| Link Model And Operation | 90 |
| Activation Spikes Along Links | 91 |

| | |
|--|-----|
| Changing Link Properties | 93 |
| Auto Link Length | 95 |
| Reassign Links to Different Neurons | 95 |
| Reassign Input Links | 96 |
| Reassign Output Links | 96 |
| General Operations | 96 |
| Display Settings | 97 |
| Customize menus, toolbars and app look | 99 |
| Using the Grid | 99 |
| Navigating In the Drawing Area | 100 |
| Selecting Objects | 101 |
| Extra Selection | 105 |
| Deleting Objects | 106 |
| Undo/Redo | 107 |
| Copy/Paste | 108 |
| Copy/Paste Activations | 108 |
| Paste to Other Applications | 108 |
| Start/Stop Simulation (Auto Think) | 109 |
| MemBrain Process Priority | 109 |
| Reset 'Don't Show Again' messages | 110 |
| CSV file separator settings | 110 |
| Standby Mode Settings | 111 |
| Reset MemBrain to default settings | 112 |
| Grouping | 112 |
| Grouping Elements | 112 |
| Adding elements to an existing group | 117 |
| Ungrouping Elements | 117 |
| Selecting Group Members | 119 |
| Changing Group Properties | 119 |
| Using Proxy Ports | 120 |
| Selecting and connecting proxy ports | 126 |
| Neural Networks | 126 |
| Net Analysis | 127 |
| Architectural Integrity | 127 |
| Layer Analysis | 129 |
| Displaying Layer Information | 132 |
| Calculating the Output | 134 |
| Resetting the Net | 135 |
| Exporting a Net | 135 |
| Net CSV File | 136 |
| Merging Nets | 137 |
| Managing I/O Data | 140 |
| The Lesson Editor | 141 |
| FFT Calculations | 148 |
| Averaging Inputs | 150 |
| Reading gray scale pixels from image files | 150 |
| Sectioned Lesson CSV File | 151 |
| Raw Lesson CSV File | 153 |
| Normalize I/O Data | 154 |
| Handling Incomplete I/O Data | 156 |

| | |
|--|-----|
| Teaching a Net | 156 |
| Selecting a Teacher/Training Algorithm | 156 |
| Randomizing the Net | 157 |
| Configure Randomization | 157 |
| Shotgun Randomization | 158 |
| The Teacher Manager | 159 |
| Teachers in MemBrain | 163 |
| Supervised Learning Algorithms | 164 |
| Standard Backpropagation | 164 |
| Backpropagation with Loopback support | 164 |
| Standard Backpropagation with Momentum | 164 |
| Backpropagation with Loopback Support and Momentum | 165 |
| RPROP with Loopback Support | 165 |
| Standard Levenberg-Marquardt | 166 |
| Cascade Correlation with Loopback Support | 167 |
| Trial and Error | 169 |
| Non-Supervised Learning Algorithms | 169 |
| Competitive with Momentum (WTA) | 169 |
| Teaching Procedure | 171 |
| The Error Graph | 172 |
| The Pattern Error Viewer | 174 |
| Auto Capturing Best Net on Stock | 175 |
| Using Group Relations to work with Sub Nets | 176 |
| What are Group Relations? | 176 |
| Adding and Editing Group Relations | 176 |
| Available Types of Group Relations | 182 |
| Choosing the currently active Relation/Sub Net | 183 |
| Convolutional Neural Networks | 184 |
| Convolutional Matrix Connection | 192 |
| Making neuron layers convolutional | 196 |
| Select Convolutions | 196 |
| Remove convolutions | 196 |
| Neural Net Stock | 196 |
| The Stock Manager | 197 |
| Net Error Calculation | 199 |
| Validating Your Net | 200 |
| Simple Validation Script | 201 |
| Launch Executable by a Neuron | 201 |
| Employ Trained Nets | 202 |
| Scripting | 202 |
| General Description | 203 |
| Executing Scripts Manually | 205 |
| The Startup Script File | 206 |
| Aborting or Suspending Script Execution | 207 |
| Remote Script Execution | 208 |
| Set Path for Remote Scripting | 209 |
| Command Line Script Execution | 209 |
| Command Line Script Compilation | 210 |
| Script Syntax | 210 |
| File Name parameters in Scripts | 211 |

| | |
|--|-----|
| Scripted Elements | 211 |
| Command Reference | 213 |
| MemBrain Application | 214 |
| Handling Neural Nets | 222 |
| Handling Group Relations | 229 |
| Editing Neural Nets | 229 |
| HandlingLessonData | 247 |
| Script class for Lessons | 264 |
| Teaching | 278 |
| Thinking | 282 |
| Adjusting the View | 284 |
| Controlling the Weblink | 285 |
| Communication with the User | 286 |
| Controlling External Applications | 289 |
| Stock Management | 292 |
| Arbitrary File Access | 296 |
| Files and Directories | 301 |
| Strings | 302 |
| String Utilities | 306 |
| Maths | 308 |
| FFT Calculations | 313 |
| Serial Ports | 314 |
| Debugging Scripts | 317 |
| Source Code Generation | 326 |
| C Code | 326 |
| Configuration | 327 |
| Generate Code | 327 |
| Build Your Application | 328 |
| C++ Code | 329 |
| Configuration | 329 |
| Generate Code | 330 |
| Build Your Application | 331 |
| The MemBrain DLL | 333 |
| Encryption | 334 |
| Set/Remove Passwords | 334 |
| Open encrypted files | 335 |
| Use Default Password | 336 |
| Check Encryption Status of Files | 337 |
| Multi Core Support / Multi Threading | 337 |
| Lesson Based Multithreading | 339 |
| Layer Based Multithreading | 339 |
| TCP Weblink | 340 |
| Operation | 341 |
| Public and Extern Neurons | 341 |
| Activate/Deactivate Weblink | 342 |
| TCP Connection Logic | 342 |
| Make Neurons Public | 343 |
| Invoke Extern Neurons | 344 |
| Make Neurons Extern | 346 |
| Extern Neuron Connection State | 347 |

| | |
|------------------------------------|-----|
| Make Neurons Private | 347 |
| Manage Active Connections | 347 |
| Remote Auto Think | 349 |
| The Nagle Algorithm | 349 |
| Protocol | 350 |
| Link Layer | 350 |
| Application Layer | 351 |
| Framing | 351 |
| Data Encoding | 352 |
| Timeouts | 352 |
| Command Reference | 353 |
| Standard Commands | 353 |
| Common Services | 354 |
| Device Capabilities | 360 |
| Commands From Public Neurons | 363 |
| Commands From Extern Neurons | 366 |
| Teacher Commands | 368 |
| Common Services | 369 |
| Commands From Public Neurons | 371 |
| Commands From Extern Neurons | 372 |
| Communication Strategies | 372 |
| Feedback and Contact | 375 |
| Used Licenses | 375 |

Welcome to MemBrain

Welcome to MemBrain

MemBrain is a powerful graphical neural network editor and simulator for Microsoft Windows, supporting artificial neural networks of arbitrary size and architecture.

Some core features of MemBrain are:

- Powerful, easy-to-learn and intuitive graphical editor and simulator for Artificial Neural Networks (ANN)
- Transfer of trained neural nets into production systems by dll or automatically generated C or C++ Code
- 'On-The-Fly'-Validation during training using separate validation data
- Supervised and unsupervised learning
- Powerful neuron and link model, time invariant or dynamic through parametrization
- Integrated high performance object oriented scripting language including graphical source level debugger
- Integrated optional 256 Bit AES encryption to help protect your IP and Know How

MemBrain originally had been developed to provide maximum flexibility for development and study of artificial neural networks combined with an intuitive graphical user interface. It is currently used by many universities for research and teachings, however, during the last years its use in industrial manufacturing and technical control applications has stepped up continuously. The majority of applications hereby is considered with the challenge to map correlations between inputs and outputs of unknown or not sufficiently known systems into artificial neural networks.

The goal of this is to be able to predict the outputs of the system under test on basis of its inputs without having to know the corresponding transfer functions. Knowledge of these functions is often not achievable at all or at least not accessible with justifiable efforts. Solely required to achieve this goal is a sufficient amount of real world input and output data pairs which might be captured from the real system or might already be available from historic records. These data can be imported into MemBrain and be used to train the corresponding artificial neural networks.

Once the neural nets have been designed and trained successfully they can be used to approximate the real system for the use in forecasts or other calculations: New values are applied to the inputs of the nets - the target output values are calculated by the nets and accessible through their output ports.

Using MemBrain, such neural nets can be easily transferred into production systems, no matter if these are PC based or embedded systems. PC based systems can use the dll version of MemBrain to load the trained nets from files and make use of them in an efficient way. The dll also allows to continue the training of the nets within the user's application.

For embedded systems MemBrain brings an integrated C as well as C++ source code generator that allows to generate code that is aligned with the typical requirements of embedded systems: Small memory footprint and avoidance of dynamic memory allocation.

Due to its flexible display properties MemBrain is applicable to big networks using many neurons and links as well as to tiny ones that only consist of a few neurons and are used for detailed examination of what happens at the lowest level.

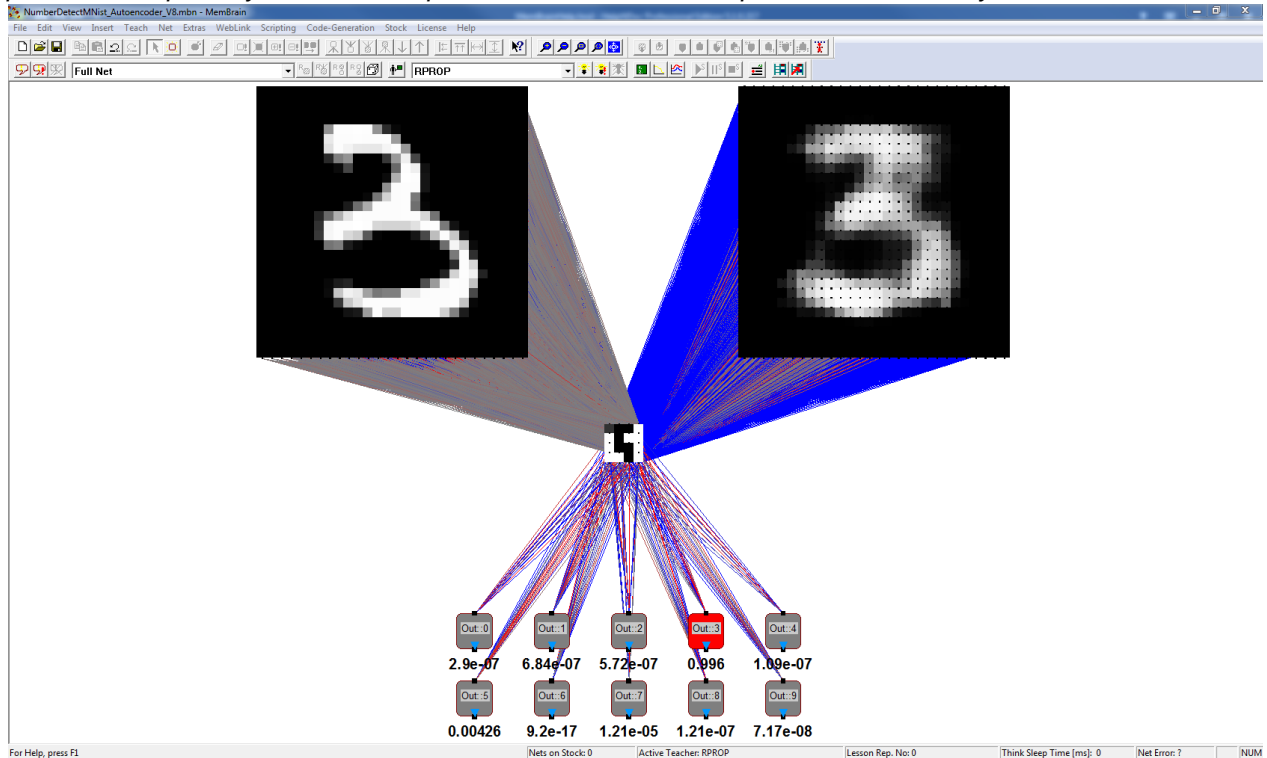
MemBrain features a powerful C++ like scripting language that allows to write everything from simple batch processes to complex control programs in order to automatically run often used sequences of operations or to achieve user defined behaviour. Scripts can be comfortably developed and debugged in MemBrain's integrated source level debugger, including setting of breakpoints, inspecting variables, step in and out of methods, etc.

When you are completely new to MemBrain the [Getting Started](#) chapter may be of help for you in order to get in touch with the most basic features of MemBrain on a simple neural net example.

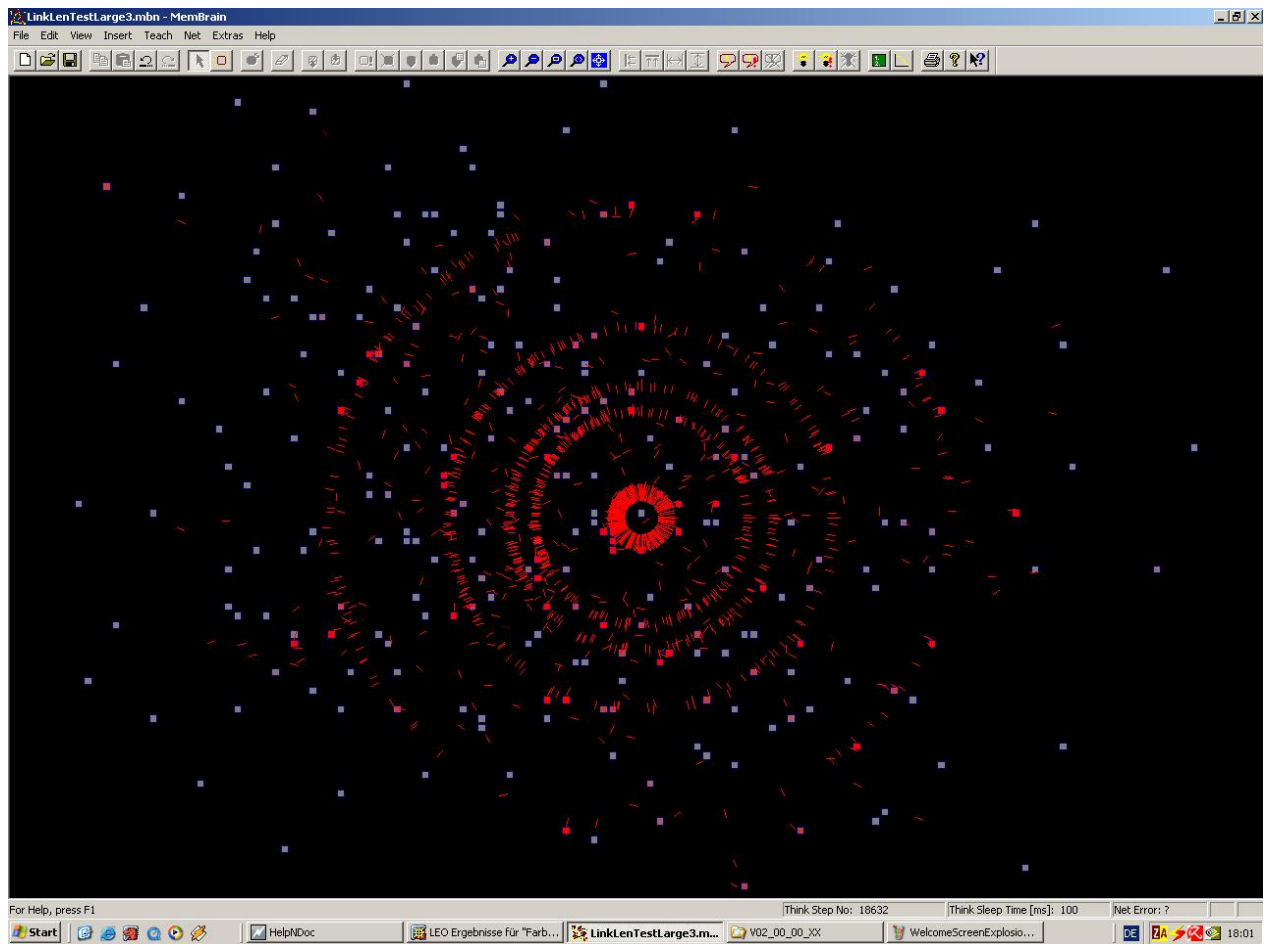
Please note that the MemBrain program and all associated supplementary software including the MemBrain DLL and the generated source code are distributed "as is". No warranty of any kind is expressed or implied. You use it at your own risk. Thomas Jetter will not be liable for data loss, damages and loss of profits or properties or any other kind of damages or loss caused by use or misuse of this software.

Find below some screen shots of MemBrain.

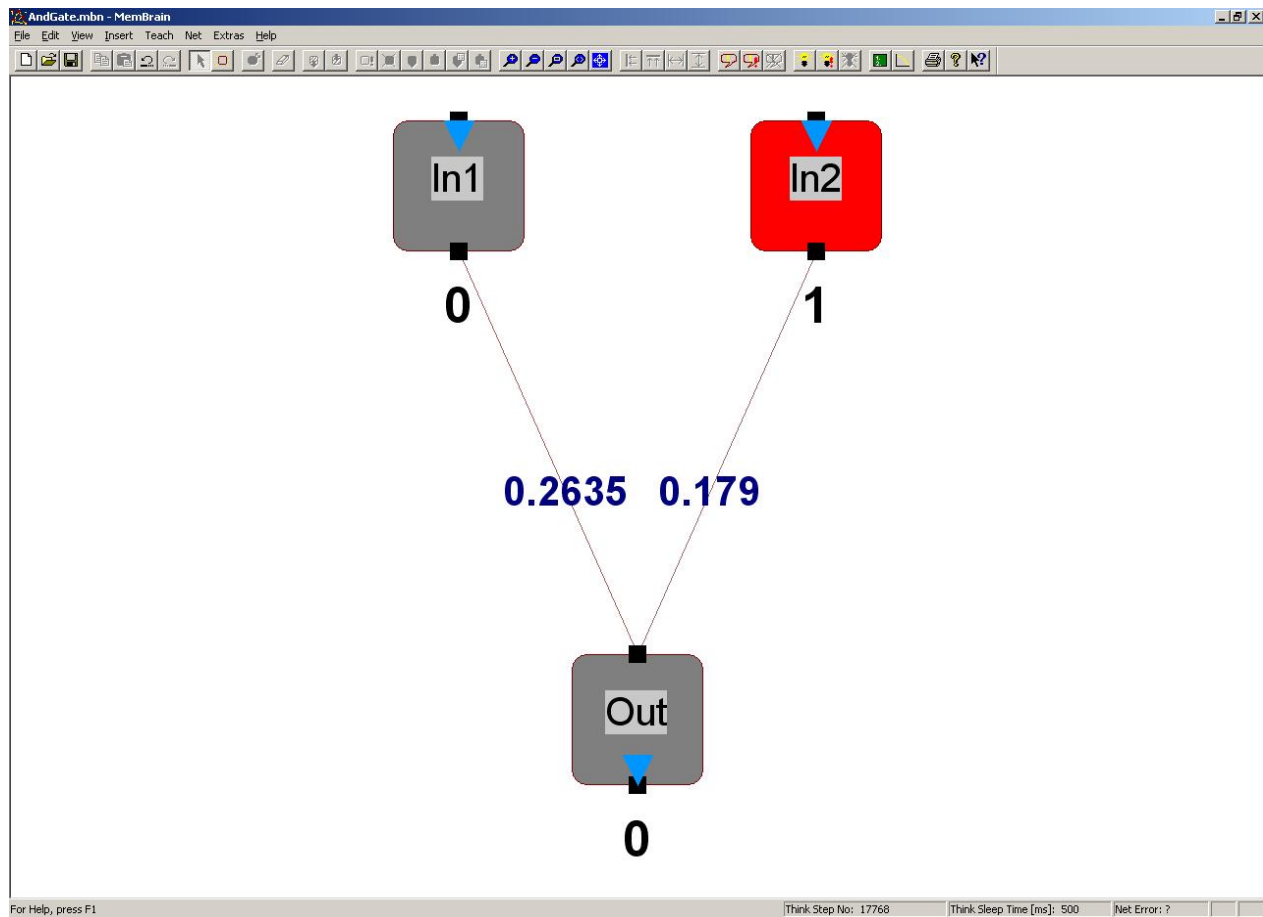
Example: Classification of handwritten digits taken from the MNIST data set. The net implements an auto encoder network on the input matrix which is pre-trained separately before the output is trained based on the pre-trained hidden layer.



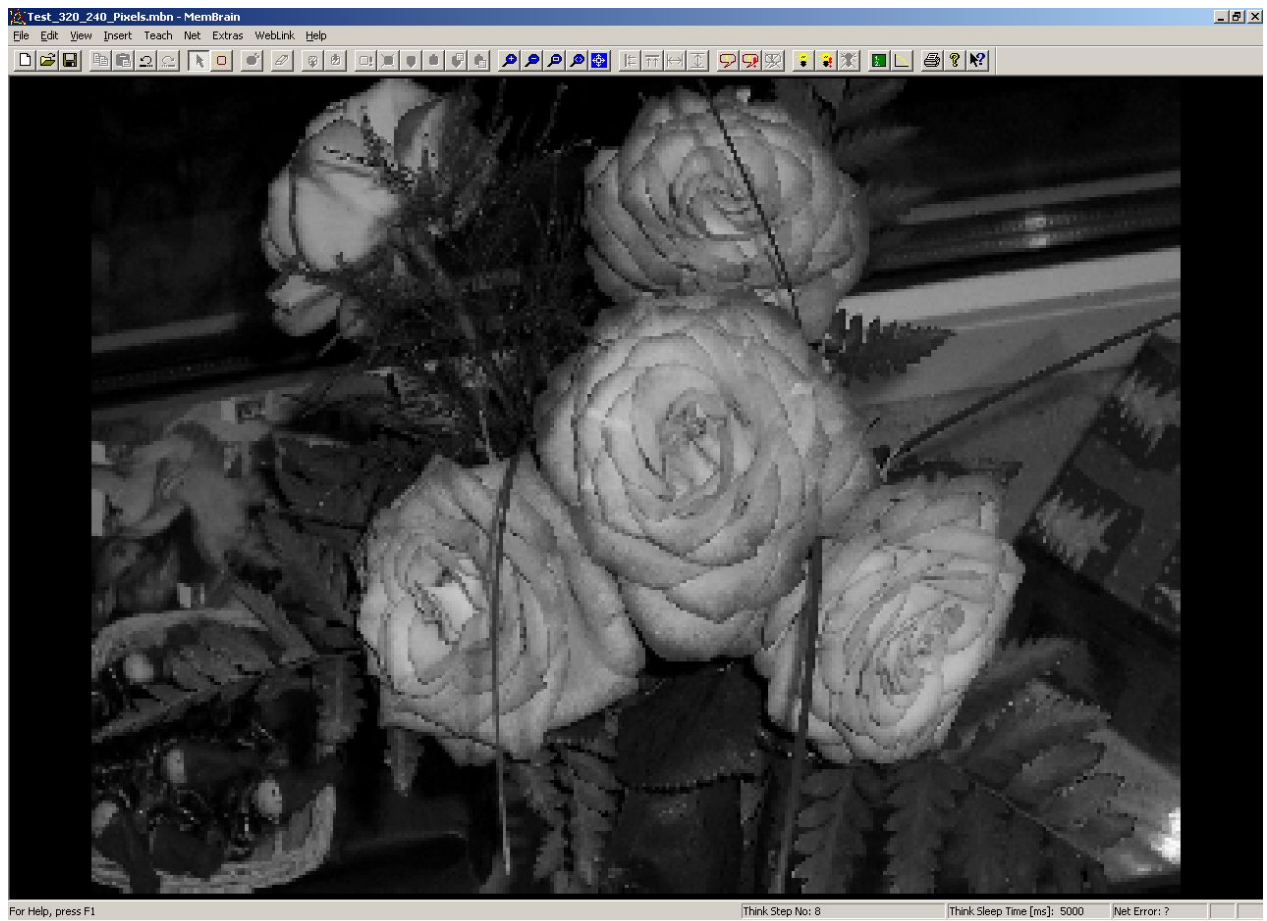
Example: Neural Net Pulsed by Spiking Neuron in its Center



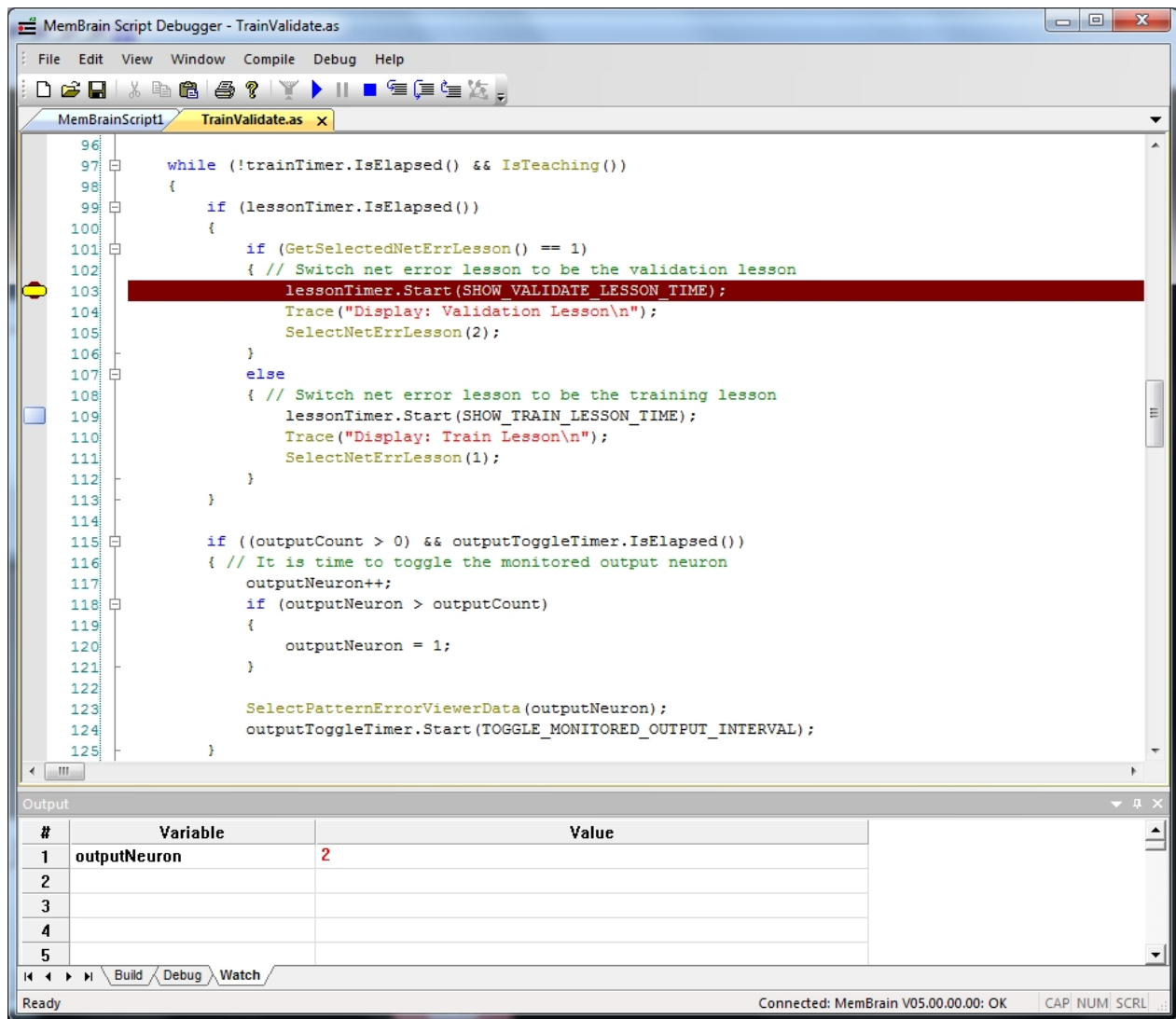
Example: Simple AND gate built with neurons



Example: A net including external neurons located on another MemBrain compatible device (here: from a driver that reads in jpeg files and publishes the pixels as neurons over TCP/IP, in this case a picture of $320 \times 240 = 76800$ external neurons)



Example: Source level script debugger



Remarks to the Help File

Important remarks to this help file:

- Some pictures in this help file have been taken from older versions of MemBrain and thus may look slightly different than they would do with the current one. E.g. the input neurons in some of the pictures have an input connector because some older version of MemBrain did allow for that. Also, the coloring of the neurons may be a bit different in some of the pictures compared to the newest version of MemBrain.
- Some screen shots in this help file show German texts on some buttons for example. This is due to the fact that this help file was created on a machine with German language. On machines with different language settings these texts will appear in the language of the machine. Most of the texts in MemBrain are hard coded in English, however, and won't change with the localization of the host computer.

Short Beginner's Tutorial

This section provides an initial "How To" on using MemBrain.

It only covers the most basic features of MemBrain and is intended for users that are completely new to MemBrain.

In this section and its sub chapters a simple Feed Forward example net is being built and trained to solve the XOR (Exclusive Or) problem.

The following figure shows the truth-table of the **XOR** function that shall be realized by the neural net.

| In 1 | In2 | Out |
|------|-----|-----|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Note that the EXOR function is not a typical application for a neural net because it is completely known (there is a simple mathematical rule to calculate the output from the inputs), it has a low number of possible input and output patterns and it is already implemented in every microprocessor of every computer.

Hence the example is very simple and thus is a good approach to get in touch with the most basic features of MemBrain.

Note that there are many advanced features in MemBrain that are not covered by this small tutorial. If you want to continue using MemBrain for your own experiments it is strongly recommended that you read through this help file, especially to learn how to efficiently use the editing functionalities of MemBrain and learn more about the used neuron and link model as well as the available learning algorithms (so called 'Teachers' in MemBrain).

If you have any questions or suggestions or whatever feedback to MemBrain then feel free to [contact](#) me, no matter if in English or German language. Any feedback is highly appreciated and will be answered!

[Next Step \(Adjust MemBrain's settings\)](#)

Adjust MemBrain's settings

First step is to make sure we have some settings adjusted in an optimal way.

This is something we only have to do once unless we want different settings again: MemBrain stores all its settings in a file when being exited and loads these settings from the file again when being launched the next time.

Please ensure the following is given for the **<View>** menu:

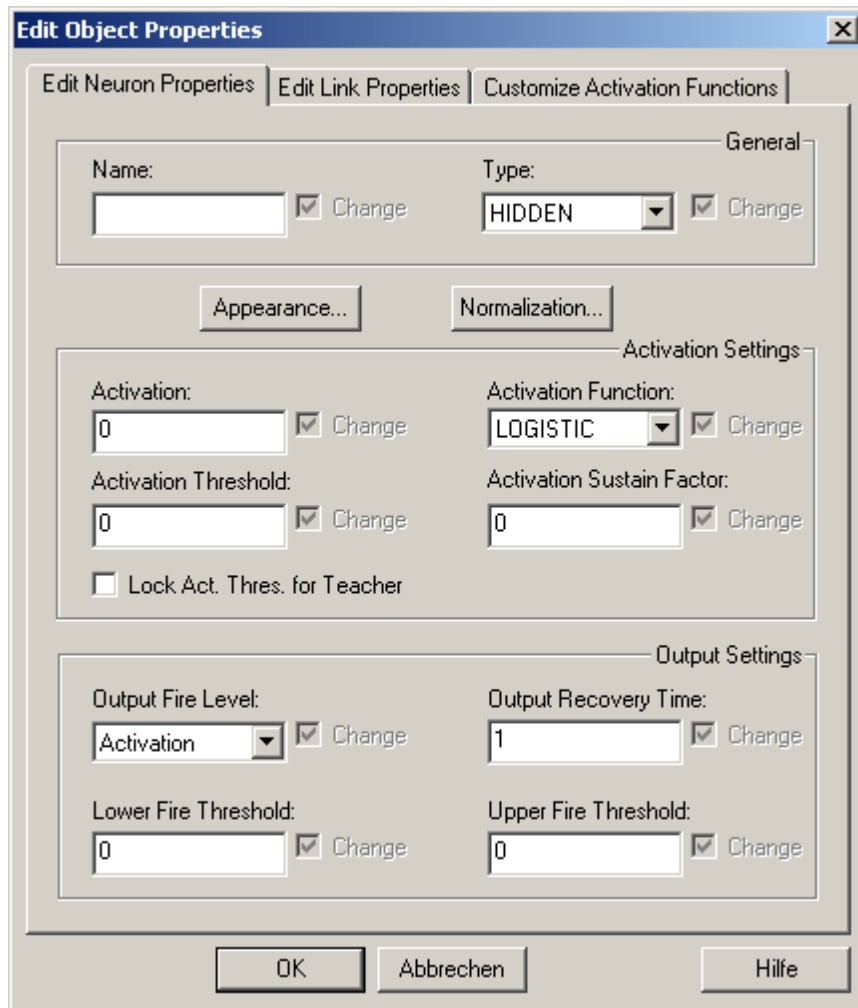
| | |
|--|--------------------|
| <View><Show Links> | : checked |
| <View><Show Activation Spikes On Links> | : unchecked |
| <View><Show Fire Indicators> | : unchecked |
| <View><Use Display Cache> | : checked |
| <View><Black Background> | : checked |
| <View><Snap To Grid> | : checked |

In the **<Teach>** menu click on the menu item **<Set Teach Speed...>**. Set 0 ms here and click **<OK>**

In the **<Edit>** menu click on **<Default Properties...>** and ensure the following is set.

Note: These should be the default settings when you start MemBrain for the first time.

On the **<Edit Neuron Properties>** tab:



Edit Object Properties

[Edit Neuron Properties](#) | [Edit Link Properties](#) | [Customize Activation Functions](#)

General

Name: ☒ Change
 Type: ☒ Change

[Appearance...](#) [Normalization...](#)

Activation Settings

Activation: ☒ Change
 Activation Function: ☒ Change

Activation Threshold: ☒ Change
 Activation Sustain Factor: ☒ Change

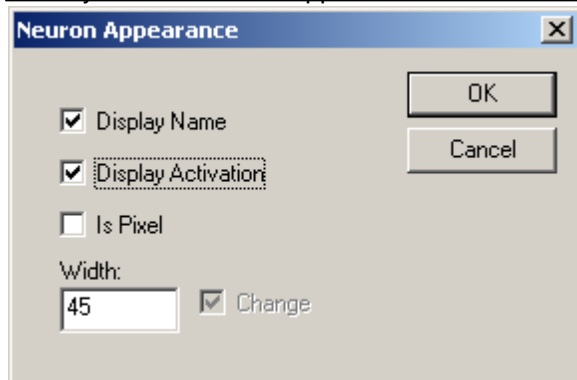
☐ Lock Act. Thres. for Teacher

Output Settings

Output Fire Level: ☒ Change
 Output Recovery Time: ☒ Change

Lower Fire Threshold: ☒ Change
 Upper Fire Threshold: ☒ Change

When you click on the <Appearance...> button on this tab:



Neuron Appearance

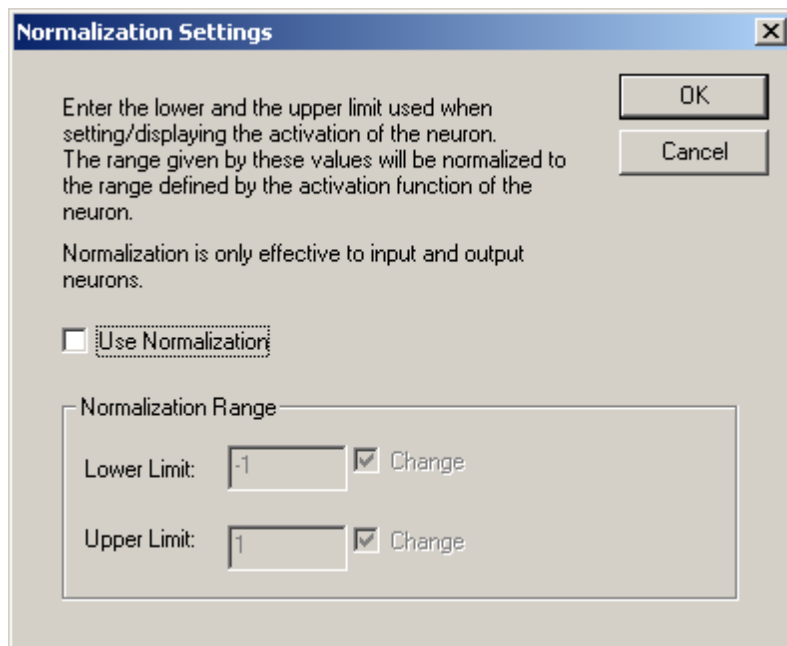
☒ Display Name

☒ Display Activation

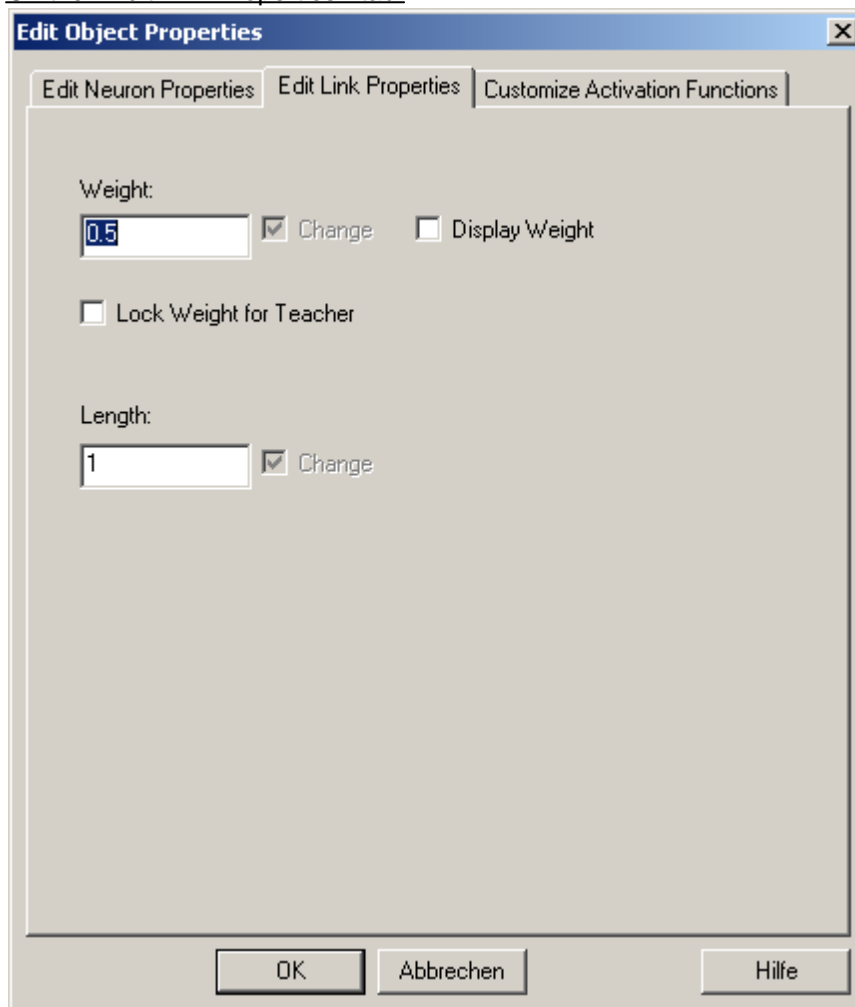
☐ Is Pixel

Width: ☒ Change

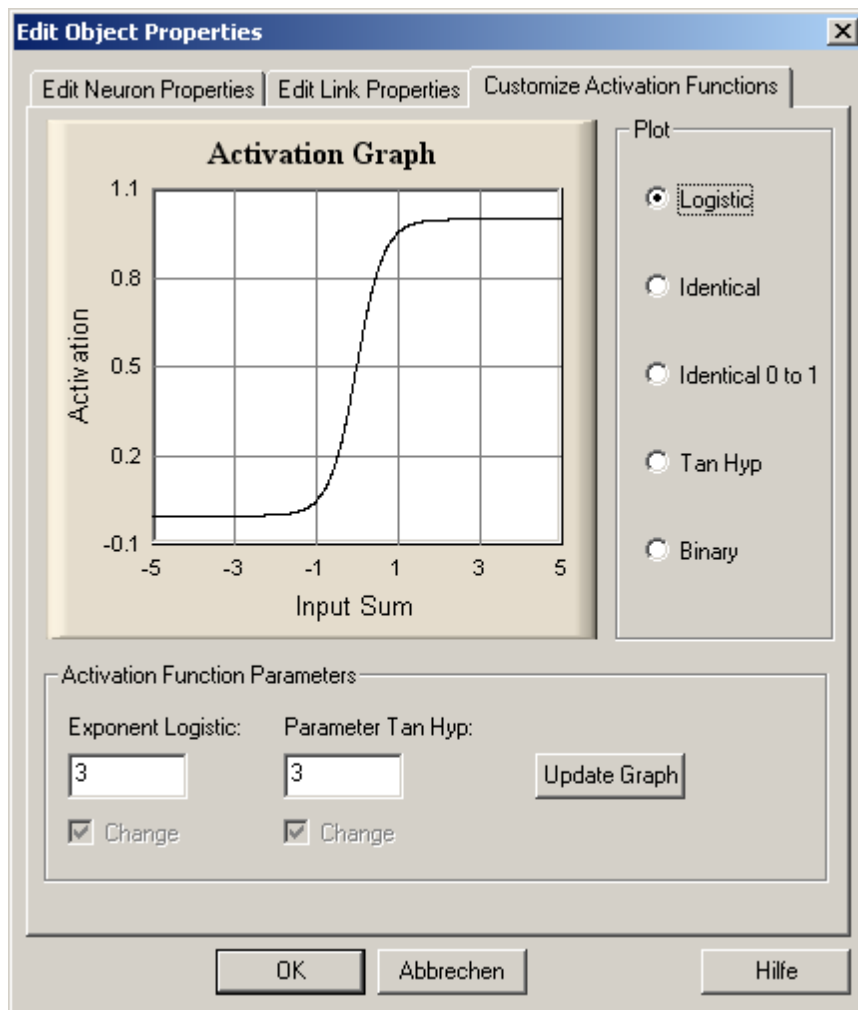
On the <Normalization...> button on this tab:



On the <Edit Link Properties> tab:



On the <Customize Activation Functions> tab:



[Next Step \(Place the neurons\)](#)

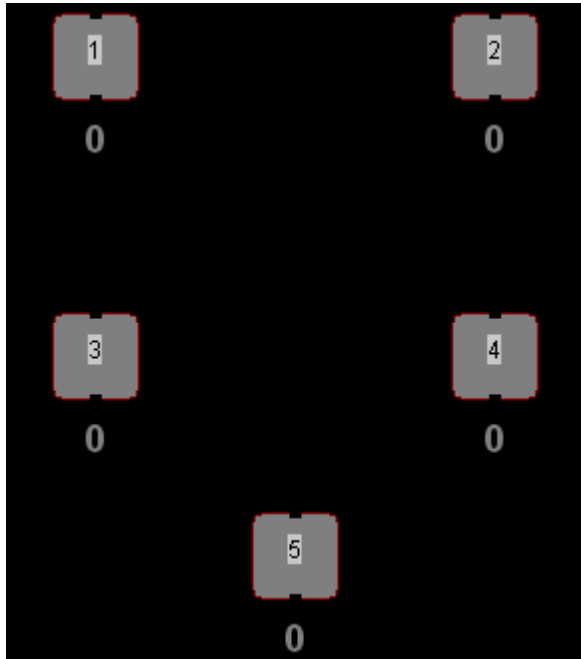
Note: If you want to learn more about what these parameters mean, click [here](#).
You won't necessarily need this knowledge for this tutorial, yet.

Place the neurons for the net

Click on the toolbar symbol



Your cursor will change to a cross hair and a new neuron will already be attached to the cursor. If you click with the left mouse button the neuron will be placed at the current cursor location and a new neuron will appear on the cursor. Continue placing neurons until you have 5 neurons arranged somehow like in the following picture.



Note: If you have placed a neuron in a wrong location don't worry, just continue placing the other neurons. You can easily move or delete the neurons later on until the net looks like you want it to. The numbers shown on the neurons may be different from here, depending on what you did in MemBrain beforehand. Don't worry about them, after all they are just default names, you don't have to care about them.

After you have placed the neurons press **<ESC>** on the key board **or** select the toolbar symbol



If you want to re-arrange some or more of the neurons just drag them across the drawing area with the mouse: Click on a neuron with the left mouse button and hold the button down. The neuron now moves with the mouse. Once you release the mouse button the neuron stops moving.

To delete neurons just select them (click on them with the left mouse button once) and press **** on your keyboard **or** select **<Edit><Delete>** from the main menu.

This may also be a good time to check out MemBrain's [Undo/Redo](#) functionality.

[Next Step \(Specify Input and Output Neurons\)](#)

Specify Input and Output Neurons

We now have to specify what the inputs and the outputs of the net are.

To do this **double click** on the single neuron on the bottom of the net. You can also **right click** on the neuron and select **<Properties>** from the context menu.

The following dialog opens.

Edit Object Properties

Edit Neuron Properties | Edit Link Properties | Customize Activation Functions

General

Name: ☒ Change

Type: ☒ Change

Activation Settings

Activation: ☒ Change

Activation Function: ☒ Change

Activation Threshold: ☒ Change

Activation Sustain Factor: ☒ Change

☐ Lock Act. Thres. for Teacher

Output Settings

Output Fire Level: ☒ Change

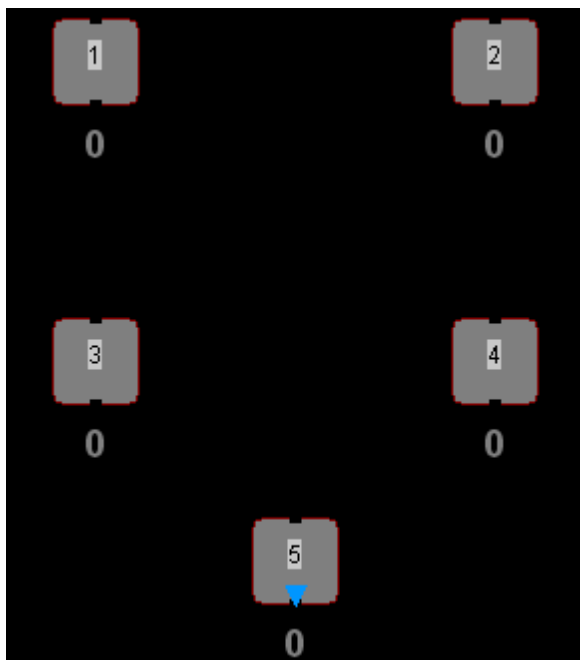
Output Recovery Time: ☒ Change

Lower Fire Threshold: ☒ Change

Upper Fire Threshold: ☒ Change

In the upper right corner of the dialog, in the field "Type" select "OUTPUT". Then click OK.

The net should now look something like this:



Note the blue arrow on the bottom of the edited neuron. It indicates that this is an output neuron. In our

example there will be only one output neuron so lets specify the input neurons now:

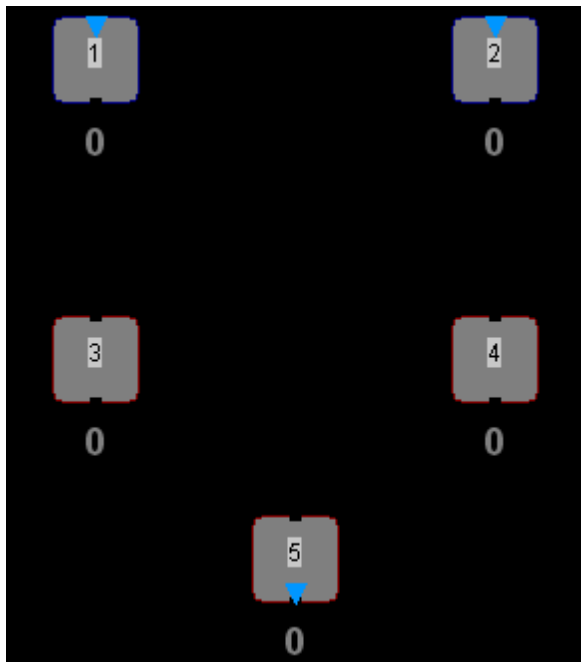
The upper two neurons shall be our input neurons. We now have to edit them both and we can perform this action for both neurons together. To do this we have to select both of the neurons:

Single click on the upper left neuron so that it gets highlighted ("selected"). Then press the **<Ctrl>** key on your keyboard, and **hold it down** while clicking on the upper right neuron. Now both neurons are selected. Alternatively you could have drawn a selection rectangle around the two neurons with the mouse. This would lead to the same result.

Press **<ENTER>** on the keyboard **or right click** on one of the selected neurons and select **<Properties>** from the main menu or **double click** on one of the selected neurons.

The above mentioned dialog will appear again. Now select "INPUT" as the neuron type and click OK.

The net now looks like this:



[Next Step \(Connect the neurons\)](#)

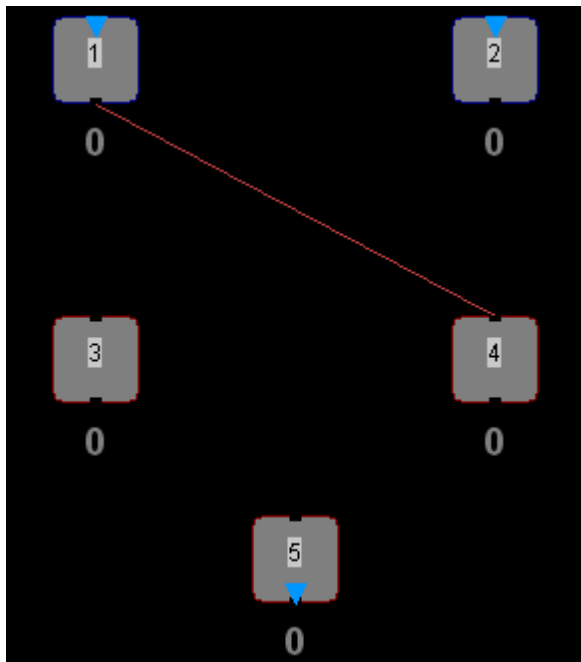
Connect the neurons

Now it's time to connect the neurons by so called links.

MemBrain provides several means to connect neurons depending on the task that shall be performed. We'll try out the simplest way first:

Hover the mouse over the little dark spot located in the middle of the lower border of the left input neuron. You will notice that once you reach this spot with the mouse it will be enlarged to a rectangular blue area. This is the output connector of the neuron. In MemBrain all neurons have their output connector on their bottom side while the input connector is located on the top side. When the output connector gets magnified (blue rectangle as described) then click the left mouse button once. You can now draw a link by moving the mouse to the input connector of the right hidden neuron until this connector gets magnified. Click the left mouse button again and the link will be connected.

The net now should look something like this:



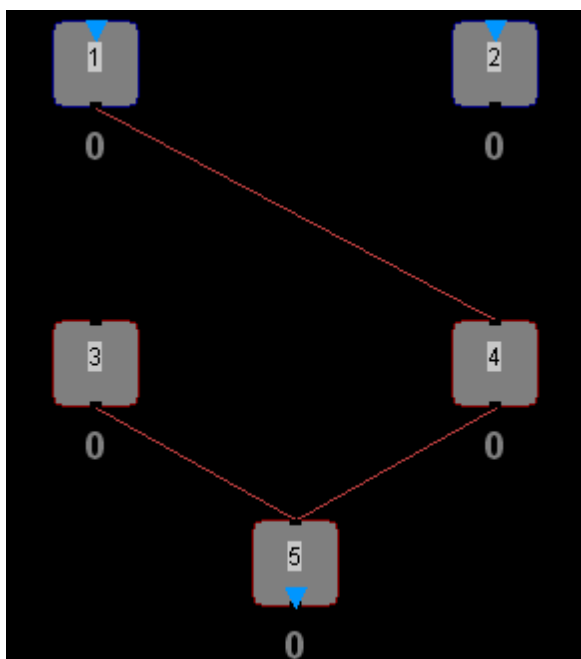
We could continue adding all other links this way but as we want to learn more about MemBrain's features we're going to use a different approach now:

Select both of the hidden neurons (draw a selection rectangle around them or select one after the other while holding down the <Ctrl> key as already performed).

Now click on the toolbar icon



You will see that you now have two links hanging on your mouse. Hover the mouse over the input connector of the output neuron (wait until it gets blue and enlarged) and click the left mouse button. This creates two new links in one shot. Notice that two further links are already generated and attached to the mouse cursor. Since we don't need them in this case just press <ESC> on the keyboard.

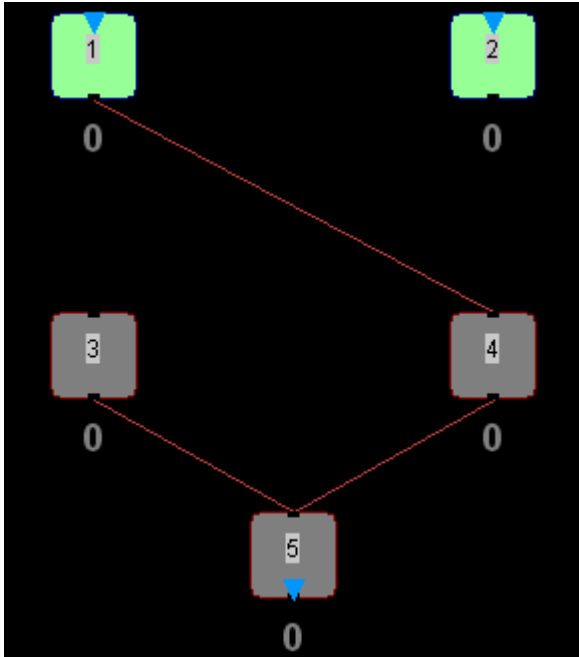


Now there are still some links missing between the input neurons and the hidden neurons. We will use another advanced method for adding these:

Select both of the two input neurons and click



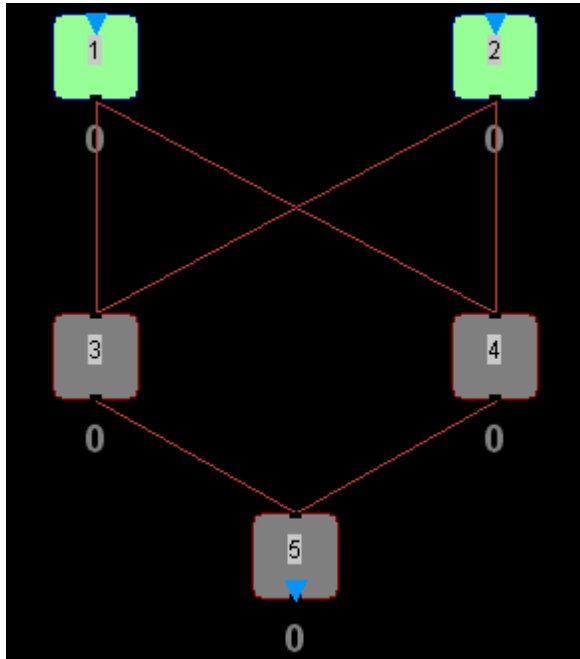
on the toolbar. The two input neurons are **"Extra Selected"** now. To see this you have to click somewhere else in the drawing area. The input neurons are colored green now:



The Extra Selection is a very important feature of MemBrain that is used for many functions that use two groups of neurons. In our example we want to connect one group of neurons (the input neurons) to another group (the hidden neurons). In order to do this we select both of the two hidden neurons and then click on the following toolbar icon.



We added all possible links with this one action:



Choose



from the toolbar to reset the Extra Selection.
Our net is almost complete now.

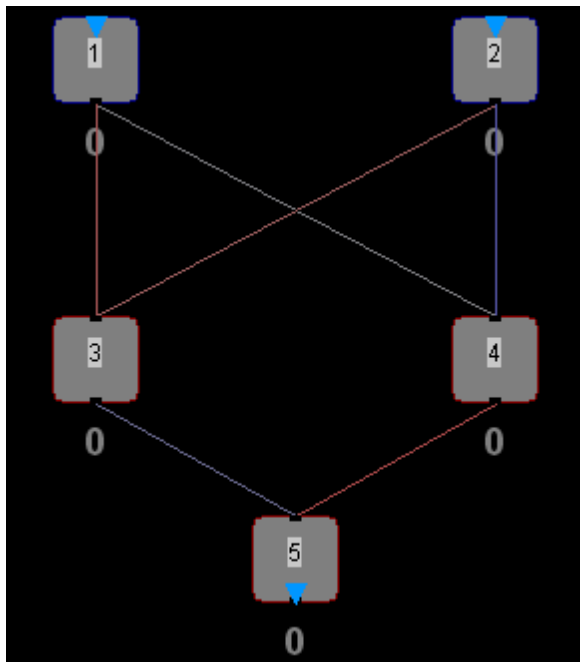
[Next Step \(Randomize the net\)](#)

Randomize the Net

Before we can teach the net in order to produce the XOR function we should randomize the weights of all links and the activation thresholds of the hidden and the output neurons of the net.

To randomize your net select **<Net><Randomize Net>** from MemBrain's main menu.

Note that the colors of the links in the net have changed to reflect their new values. The net could look something like this now:



If you double click on one of the hidden neurons you will see the following dialog.

Edit Object Properties

☒ Edit Neuron Properties
 ☐ Edit Link Properties
 ☐ Customize Activation Functions

General

Name: ☒ Change
 Type: **HIDDEN** ☒ Change

Activation Settings

Activation: ☒ Change
 Activation Function: **LOGISTIC** ☒ Change

Activation Threshold: ☒ Change
 Activation Sustain Factor: ☒ Change

☐ Lock Act. Thres. for Teacher

Output Settings

Output Fire Level: **Activation** ☒ Change
 Output Recovery Time: ☒ Change

Lower Fire Threshold: ☒ Change
 Upper Fire Threshold: ☒ Change

Note the value for the activation threshold: This has been set by the randomize function to some small positive or negative value. That happened to all neurons of the net except for the input neurons where this

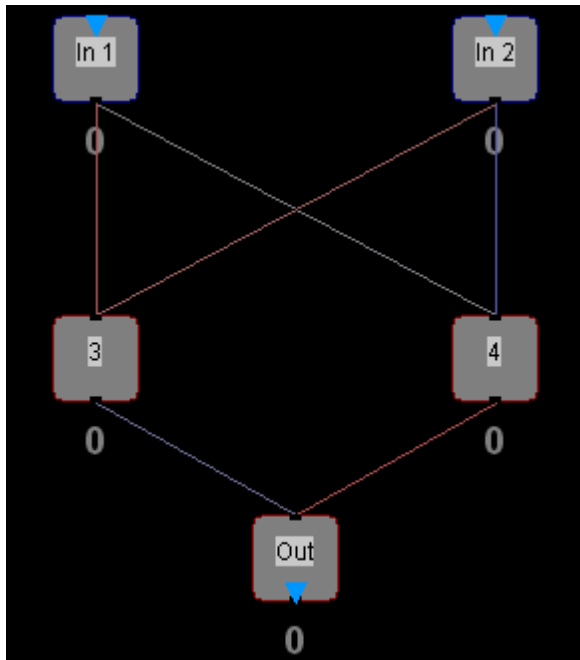
threshold is not used.

If you want to learn more about MemBrain's neuron model, see [here](#).

[Next Step \(Rename the Neurons\)](#)

Rename the Neurons

To give the input and output neurons a bit more meaningful names **double click** on them and assign them new names in the **properties dialog** so you get something like this:



Now we are finished with editing our simple neural net. This is a good time to save the net. (Main menu: **<File><Save>**)

[Next Step \(Enter I/O Data\)](#)

Enter I/O Data

In order to train the net for the XOR function we have to create the input/output (I/O) data sets that make up this function.

I/O data in MemBrain is handled by the Lesson Editor.

Click the toolbar symbol



to open it.

If you open the Lesson Editor for the first time after you have started MemBrain it automatically adjusts to the I/O neurons of your net. If you have changed I/O neurons or their names **after** opening the Lesson Editor you first have to click on the Lesson Editor's button **<Names From Net>** and confirm the message box that

opens up then.

You now should see the following dialog.

Lesson Editor: unnamed

Lesson Files MemBrain CSV Files Raw CSV Files Extras

Number of Lessons: 1 Currently Edited (Training) Lesson: 1 Net Error Lesson: 1

☐ Set Manually

Input data:

| In 1 | In 2 |
|------|------|
| 0 | 0 |

Pattern Name: New Pattern Pattern No: 1 of 1 Comment

☒ Output Data:

| Out |
|-----|
| 0 |

Sync With Net

Names from Net Number of Inputs: 2 Apply Edit/Lock Names

Names to Net Number of Outputs: 1 Apply

New Pattern Delete Pattern Delete All Patterns

Data to Net

Think on Input Think on Next Input

Think on Lesson

Data from Net

Pattern from Net New Pattern from Net

Record Lesson

☐ Record one pattern every

1 Think Step

To Lesson No. 1

☒ Activations ☐ Outputs

Name of Lesson: New Lesson Comment

Close

Note that the section **"Input data"** contains the names of the two input neurons of our net. The section **"Output data"** lists the output neuron we added to the net.

In between the two areas you see the note **"Pattern No: 1 of 1"**. This indicates that we currently have one I/O pattern in our data set or, to use MemBrain terminology, in our "Lesson".

Remember that we need 4 patterns to cover the functionality of the XOR function. Thus click on the button **<New Pattern>** now **three times**. The mentioned statement should have changed to **"Pattern No: 4 of 4"** now. You can scroll through all patterns of the lesson by using the **Up/Down arrow buttons** on the right side of the Lesson Editor. By default all pattern data is 0, so pattern No 1 is already OK (In 1 = 0, In 2 = 0, Out = 0).

Select pattern No 2 now and enter the following data into the cells of the sections "Input data" and "Output data":

In 1 = 0, In 2 = 1, Out = 1

Do the same for pattern 3 and 4 with the following data:

In 1 = 1, In 2 = 0, Out = 1

and

In 1 = 1, In 2 = 1, Out = 0

respectively.

Now, from the Lesson Editor menu, select **<Lesson Files><Save Current Lesson>** and store the created lesson somewhere on your hard disk. A good name would be "ExOrGate.mbl" for example.

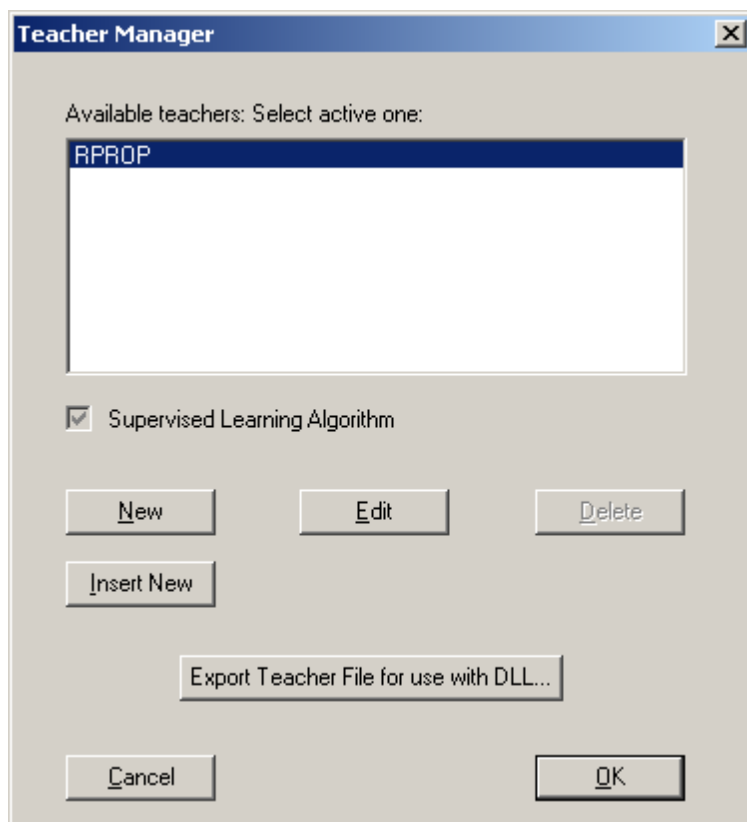
You can close the Lesson Editor now if you like to but you don't have to: The entered data keeps being active no matter if the Lesson Editor is open or not.

[Next Step \(Configure a Teacher\)](#)

Configure a Teacher

Now we will configure a teacher (i.e. learning algorithm) for our net:

Choose **<Teach><Teacher Manager...>** from MemBrain's main menu. The Teacher Manager opens as following.



In the list of available teachers you see one entry only. This is the default teacher. We're now going to create a new teacher and configure it for our task:

Click on the button **<New>**. This will bring you to the following dialog:

Edit Teacher

Name:

Type:

☒ Supervised Learning Algorithm

Learning Rate: Repetitions per Lesson: Repetitions per Pattern:

Target Net Error (for Auto Teacher):

☐ Online Learning (Batch Learning if not checked)

☒ Use Lesson ☒ Re-Apply Pattern (when repeating Patterns)

☒ Wait for Weblink Data Reception

☒ Use Weblink Sync

☐ Enable Weblink Remote Control

☐ Reset Net Before Every Lesson

☐ Rename Winner Neurons According to Patterns

☒ Use On-The-Fly Net Error Calculation if possible

Lesson Pattern Selection

☐ Ordered

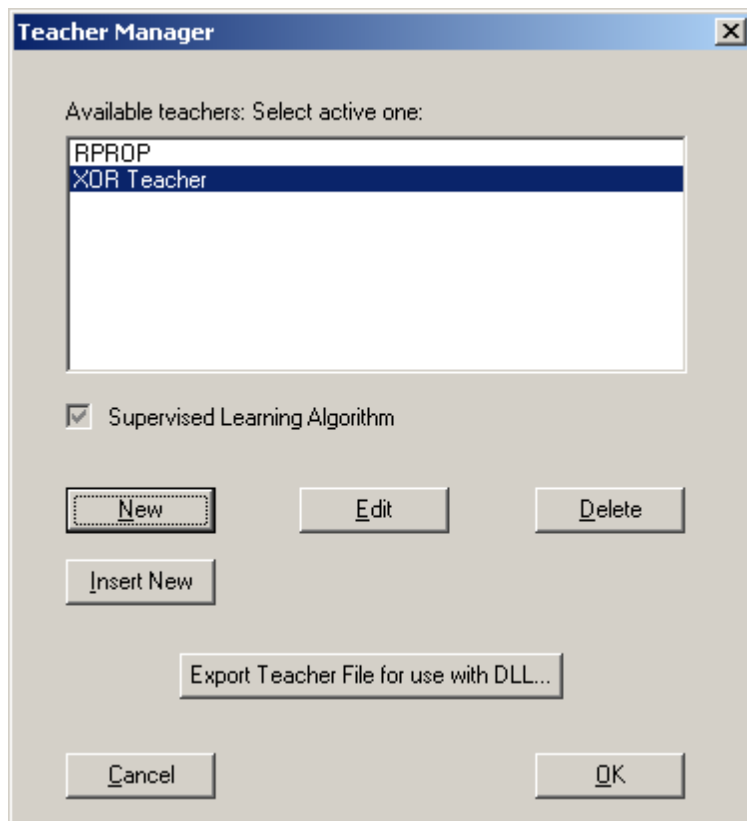
☐ Random Selection

☒ Random Order

Type "XOR Teacher" in the "Name" edit box on the top of the dialog.

The parameters are already adjusted in an appropriate way, we do not have to make any edits here.

You will be taken back to the Teacher Manager:



Note that our new teacher is set as the active one automatically. It is important to know that the teacher that is currently selected in the Teacher Manager is the one that will be used for teaching!

Close the Teacher Manager now by clicking **<OK>**.

[Next Step \(Teach the Net\)](#)

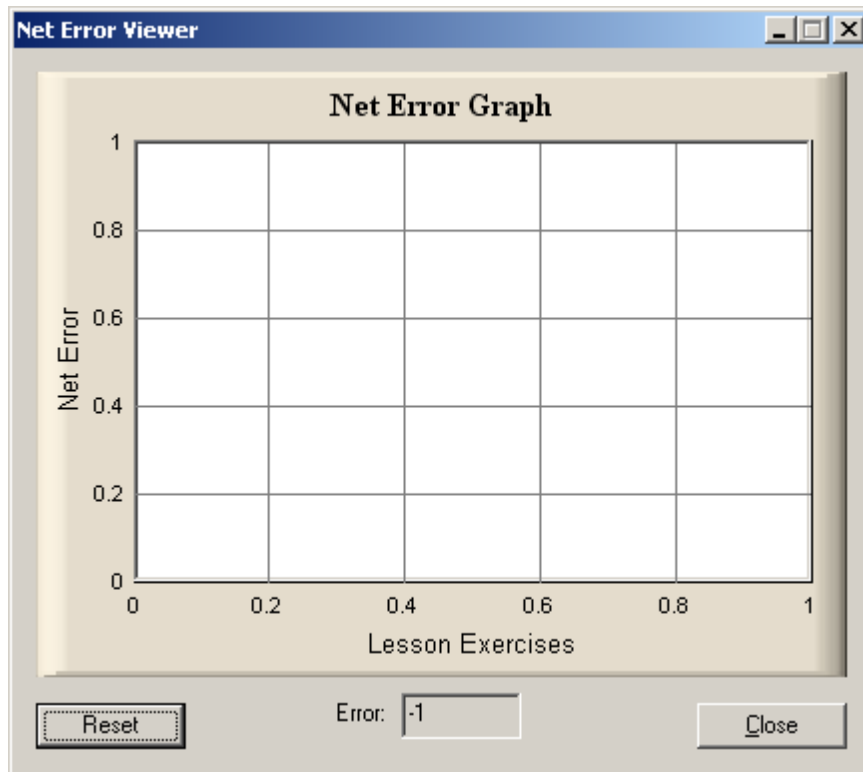
Teach the Net

Now we are ready to teach our net.

In order to be able to view the progress of the teaching process click on the toolbar symbol



The net error viewer will open:



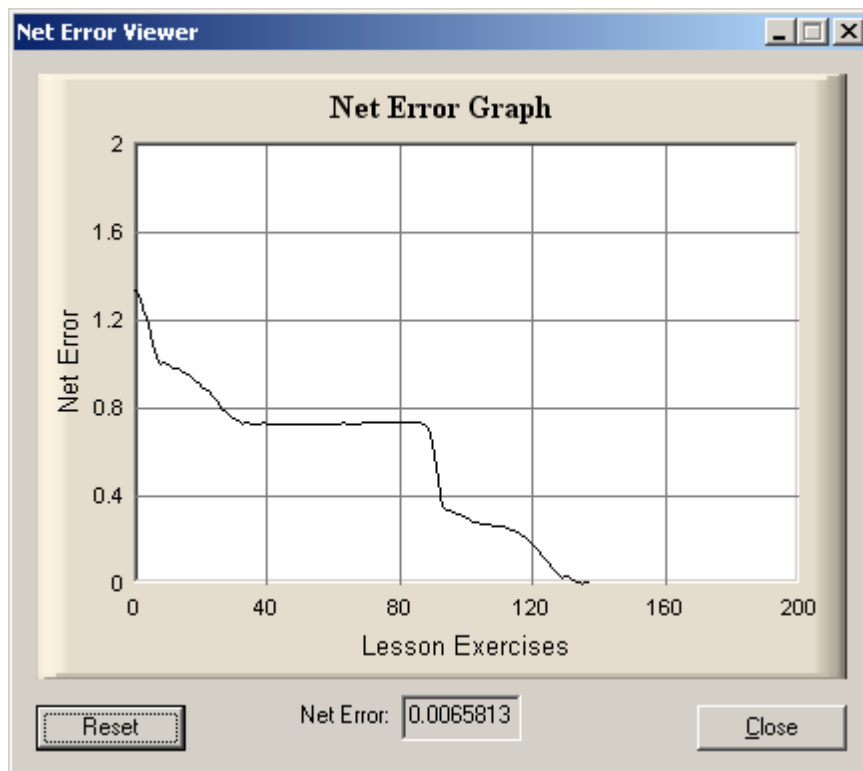
You can arrange your display so that you see both the Error Viewer and your complete neural net: Hold down the **<SHIFT>** key on your keyboard. You will see that the mouse cursor changes to some fourfold arrow. Then click with the **left mouse button** somewhere into the drawing area and **hold the mouse button down**. Move the mouse while the left mouse button is down and you will see that you can **drag the whole drawing area**. Actually what you do with this is moving the visible rectangle of your drawing area.

By moving the visible area and the Error Viewer window you can arrange for a comfortable display. Note that you can also [zoom in and out](#) in order to best fit your requirements.

Now start the teacher by clicking on the following toolbar symbol.



After a few seconds your Net Error Viewer could look something like this:



Note: The graph you see probably will look different from that because through the randomization we performed on the net your net has different starting conditions.

Stop the teacher now by clicking on



The net is trained now (the error is close to zero as we can see in the Net Error Viewer). Note that the net error is also displayed on the right side of MemBrain's status bar on the bottom of the window. For more information on the meaning of the net error see the chapter on the [Error Graph Viewer](#).

[Next Step \(Check Patterns\)](#)

Check Patterns

We will now check if the trained net correctly reproduces the patterns of the XOR function.

Open the Lesson Editor again by clicking on



in the toolbar.

In the Lesson Editor select Pattern 1 of 4 with the **Up/Down** arrows. Then click on the button **<Think On Input>**. This applies the input data of the active pattern to the net and triggers one calculation step of the net.

The output neuron should now have an activation close to 0 as the input pattern 0/0 is applied.

Check out the output of the net to all other patterns by repetitively clicking on the button **<Think On Next Input>**.

The activation of the output neuron should always be close to the corresponding target value in the Lesson Editor. You can continue teaching the net if you want to have better results.

Instead of using the Lesson Editor you can also assign activations manually to the input neurons and then click on the toolbar symbol



to cause the net to perform a calculation step (to [calculate the output](#))

To assign activations to neurons there are two different options:

- Double click on a neuron and enter an activation value manually on the properties dialog
- Use the [Quick Activation](#) feature

You may also want to check out the reaction of the net to data other than 0 or 1. Just go ahead and try! How does the net react on values like 0.1 or 0.9 for example?

If the reaction on this kind of data is not satisfactory you can also add these patterns to the Lesson in the Lesson Editor and teach the net again including these patterns. Does the net behaviour improve?

[Next Step \(Examine the Net\)](#)

Examine the Net

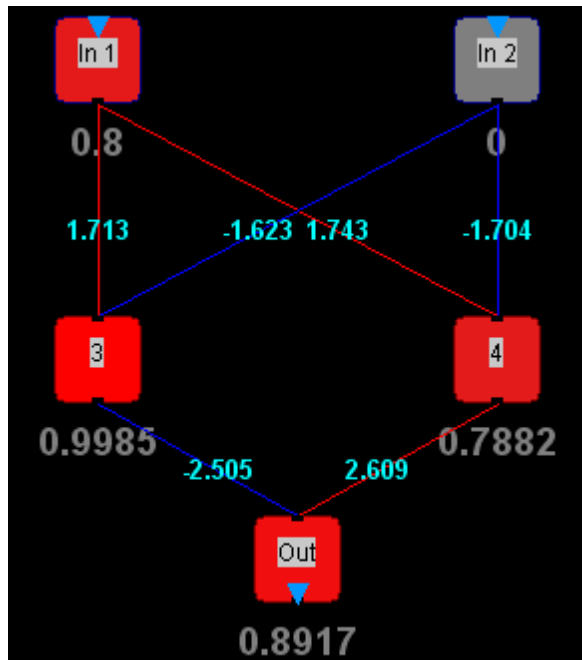
Now that the net has learned to reproduce the XOR function you might want to know how it did that?

It would blast the scope of this help file to provide basics about neural nets in general. The answer lies in the weights of the links between the neurons and the activation thresholds of the neurons.

To visualize the weights of all links do the following:

- Press **<CTRL> + 'A'** on the keyboard.
- Hit **<ENTER>** on the keyboard.
- Switch to the **<Edit Link Properties>** tab of the dialog that appears.
- Place a check mark in the field **<Display Weight>**
- Click on **<OK>**
- Click somewhere in the drawing area to reset the selection

Now the weights of all links are displayed on the links themselves:



The **activation thresholds** of the neurons cannot be visualized directly, you have to **double click** on a **neuron** and look up its activation threshold in the properties dialog that opens.

To get further information about your first neural net you can also use the [Network Analysis](#) and/or MemBrain's capability to [display Layer Information](#).

This is the end of the short "Getting Started Tutorial".

Note that this tutorial only mentions some very basic features of MemBrain. If you want to go deeper with neural nets and MemBrain you are strongly recommended to read through the rest of the help manual. You will also learn then how to deal with bigger neural nets and many more data patterns, as well as how to use a neural net productively - certainly, typing in activation values of single neurons is nothing practicable for real life use of a neural net.

Also, if you want to use the dynamic, time dependent simulation capabilities of MemBrain or if you want to work with recurrent networks or even distributed networks you will need further information from this help file and also get in touch with the neuron model of MemBrain and its parameters.

Note that there are some [examples](#) provided with MemBrain that might be a good inspiration for further experiments.

Included Examples

Some examples are provided together with MemBrain. They can be downloaded as a package from the [MemBrain homepage](#).

The package includes a small help file with some information on the examples. Furthermore there are scripting examples available on the homepage which show how scripting can be used in MemBrain.

Neurons in MemBrain

Neurons in MemBrain are displayed as following.

Input Neuron:



Hidden Neuron:



Output Neuron:



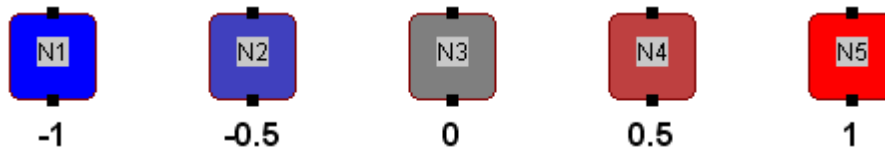
Neuron with locked activation threshold (one of the [properties](#) of a neuron) indicated by its blue border line. A locked activation threshold of a neuron will not be changed through teaching.



Generally in MemBrain a neuron's input port is on the top of the neuron, the output port is located at its bottom. Note that the option <View><Show Links> must be activated in order to see the ports.

N1, N2, N3 and N4 are the names of the neurons. Every neuron can be assigned a free selectable name. This is one of the properties of the neurons.

Neurons will change their color according to their current activation: A neuron with an activation of 0 will be displayed in grey color. A negative activation will cause the neuron to be displayed more in the direction of blue, reaching its maximum blue coloring at an activation of -1. Positive values of the activation result in a red colored neuron. The maximum red coloring will be reached with an activation of 1:



Different colors indicate different activations of the neurons

Note that the color always is determined on basis of the so called 'normalized' activation of the neuron. User defined normalization ranges can be applied to input and output neurons as described [here](#).

If you want to learn more about MemBrain's mathematical neuron model then click [here](#).

See [Changing Neuron Properties](#) for details on how to modify the properties of one or more neuron(s).

Adding Single Neurons

To add new neurons to the drawing area just select <Insert><New Neurons> from the main menu or right-click somewhere into the free drawing area and choose <Add Single Neurons> or simply click on the corresponding tool bar button:



The cursor will change to a cross hair and a new neuron will already be attached to the cursor. Move the cursor to the designated position and place the neuron at the cursor position by clicking the left mouse button. The next new neuron will already be attached to the cursor. Continue placing new neurons the same way. When finished select the standard operation mode again by clicking



on the tool bar, by de-selecting <Insert><New Neurons> from the main menu or by simply hitting <ESC> on the keyboard.


Note:

While adding neurons you can also [navigate](#) through the drawing area. If the [Snap to Grid](#) option is activated, placement of neurons will automatically happen at grid points only. In this case be sure to have the grid width adjusted to a value suitable for your planned neural net.

New neurons are always created with the current default properties set by <Edit><Default Properties...> from the main menu. The name of the new neurons will consist of the string entered for the name in the default properties plus a <SPACE> character and a unique number that is created internally by MemBrain to identify each single neuron. This default name can be modified as desired. Changing the name will not influence the internal MemBrain identifier for a neuron.

For more information about neurons in MemBrain click [here](#).

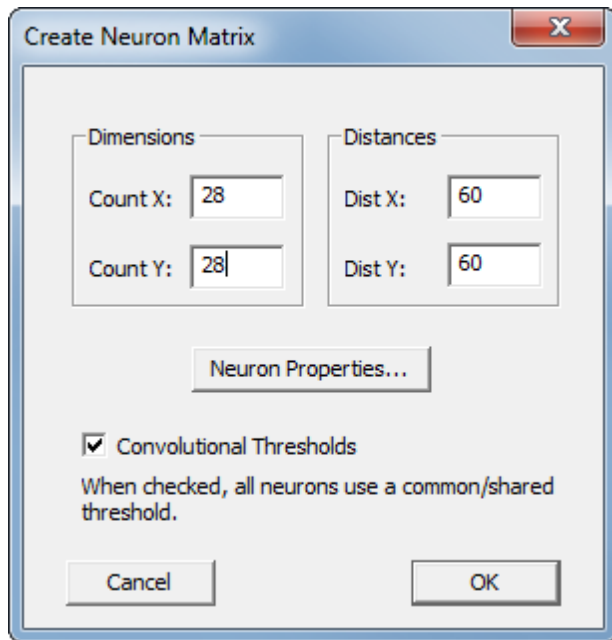
Adding Neuron Matrices

To add a matrix of neurons to the net select <Insert><Neuron Matrix...> or click on the toolbar button  or right-click somewhere into the free drawing area and choose <Add Neuron Matrix...>.


The following dialog appears and lets you configure the new neuron matrix.

You can adjust the number of neurons in X and Y dimension as well as their placement distances. The button <Neuron Properties...> allows to adjust all aspects of the inserted neurons (e.g. type, activation function...).

Check the box <Convolutional Thresholds> if you want to have all the newly created neurons share the same threshold value. This is an approach to reduce the number of free parameters of the neural network and is typically used in combination with shared link weights to form convolutional layers.

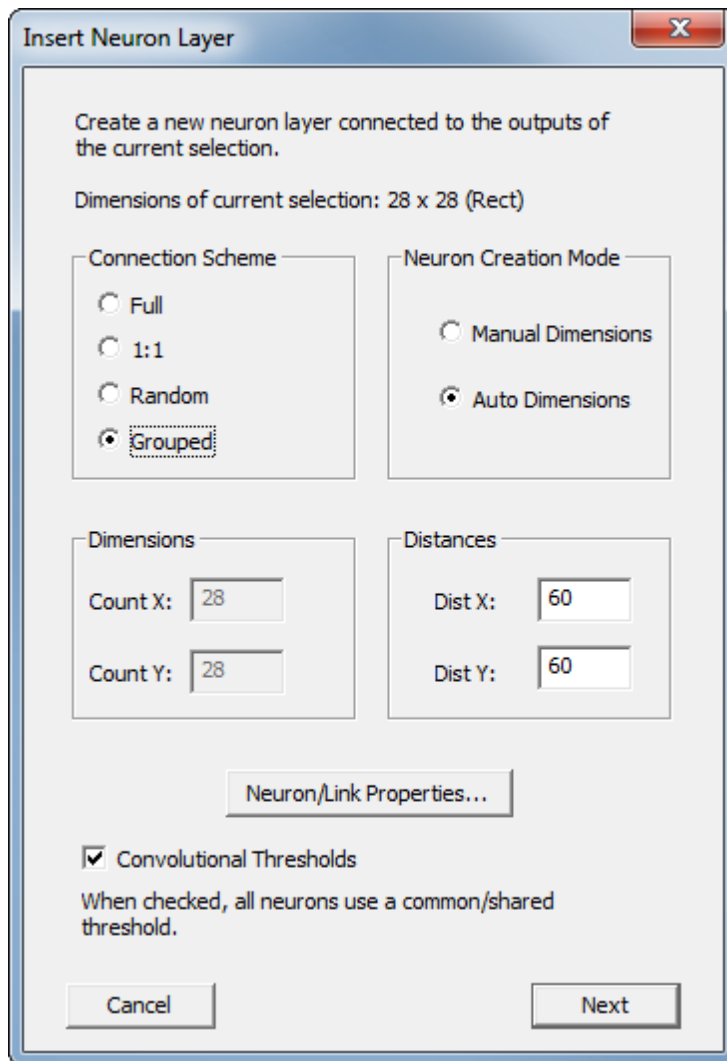


Adding Neuron Layers

In order to add a new matrix-shaped layer of neurons, connected to the outputs of the currently selected neurons, select <Insert><Neuron Layer...> or click on the toolbar button  or right-click on one of the currently selected neurons and choose <Add Neuron Layer...>.

The following dialog appears and lets you configure the new neuron layer.

You can adjust the number of neurons in X and Y dimension as well as their placement distances. The button <Neuron/Link Properties...> allows to adjust all aspects of the inserted neurons and links (e.g. type, activation function...).



Depending on the selected connection scheme the button in the lower right corner of the dialog either shows the text <OK> or <Next>.

When <Next> is shown, like in the example above, there is more information required. Clicking <Next> will bring up another data entry window. For the above example this will be:

Layer Creation and related Matrix Connection Specification

Specify below how the sub groups shall be chosen to interconnect the current selection and the newly created following layer

The connections will be arranged using rectangular neuron groups with the specified dimensions in neurons. Additionally you can specify if the groups shall overlap each other and if so, then how big the overlap area shall be.

Selection Grouping - Matrix dimensions = 28 x 28 neurons

Group X Size: Group Y Size:

☒ No Group Overlap
☐ Overlap Groups By: Neurons

The current settings result in 784 connection groups for this matrix

Connect to:

New Layer Grouping - Matrix dimensions = 28 x 28 neurons (auto-size)

Group X Size: Group Y Size:

☒ No Group Overlap
☐ Overlap Groups By: Neurons

The current settings result in 784 connection groups for this matrix

☒ **Convolutional weights**

The matrix connection is feasible.

This dialog lets you specify the connection scheme used for connecting to the newly created layer (compare also when connecting already existing neuron matrices, described [here](#)).

Since in the first dialog the <Auto Dimensions> option was selected the dialog above automatically recalculates the size of the newly created target layer when input data in its edit fields changes.

Check the box <Convolutional Thresholds> if you want to have all the newly created neurons share the same threshold value. This is an approach to reduce the number of free parameters of the neural network and is typically used in combination with shared link weights to form convolutional layers.

Clicking on OK will insert the new layer, connected to the outputs of the currently selected neurons in the specified way.

Adding Decay Neurons

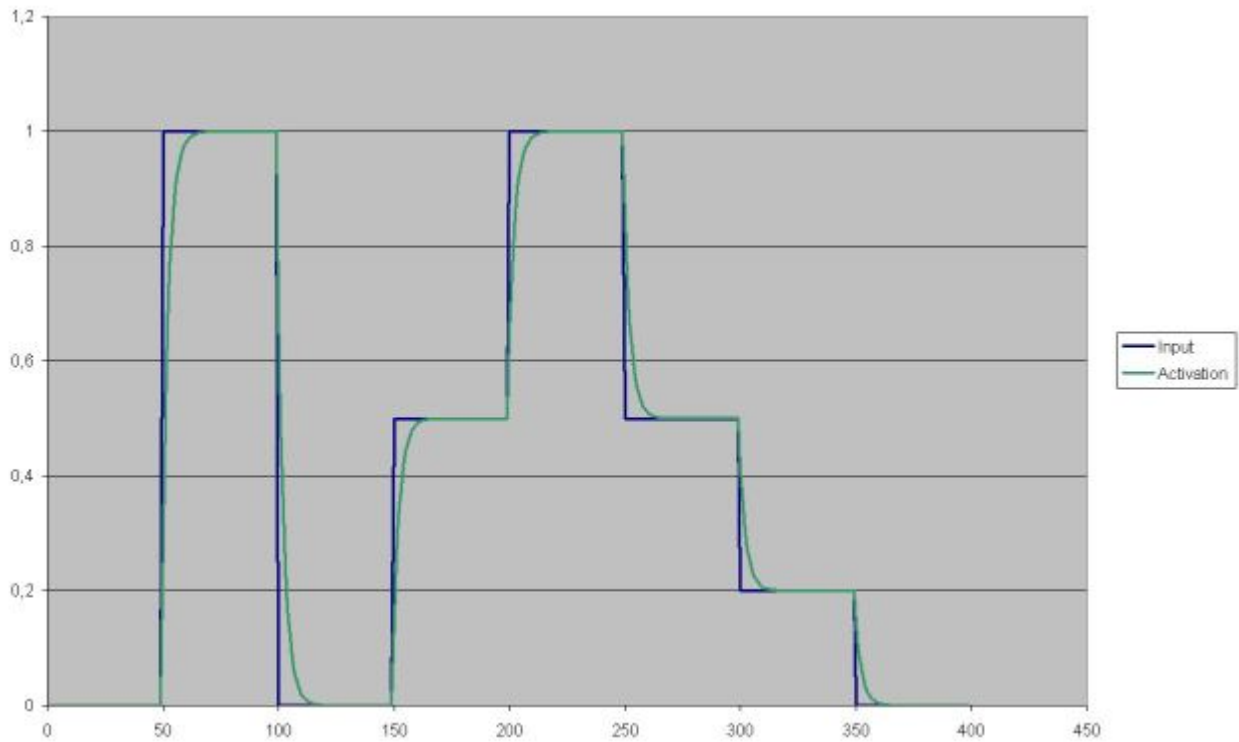
If you want to build a time variant net in MemBrain one option is to add so called decay neurons to your net.

A decay neuron is a hidden neuron whose 'Activation Sustain Factor' is set to a value > 0. This means that

this neuron incorporates a certain portion of its current activation into the calculation of its next activation.

Assume that the Activation Sustain Factor of a neuron has the value k . If this neuron gets supplied with input values over a link of the weight $(1 - k)$ then the neuron is properly configured to build a decay neuron which shows some kind of low pass filter characteristic towards signals entering its input.

The following chart shows the step function response of a decay neuron that has an Activation Sustain Factor of 0.3 and a corresponding input link weight of 0.7.



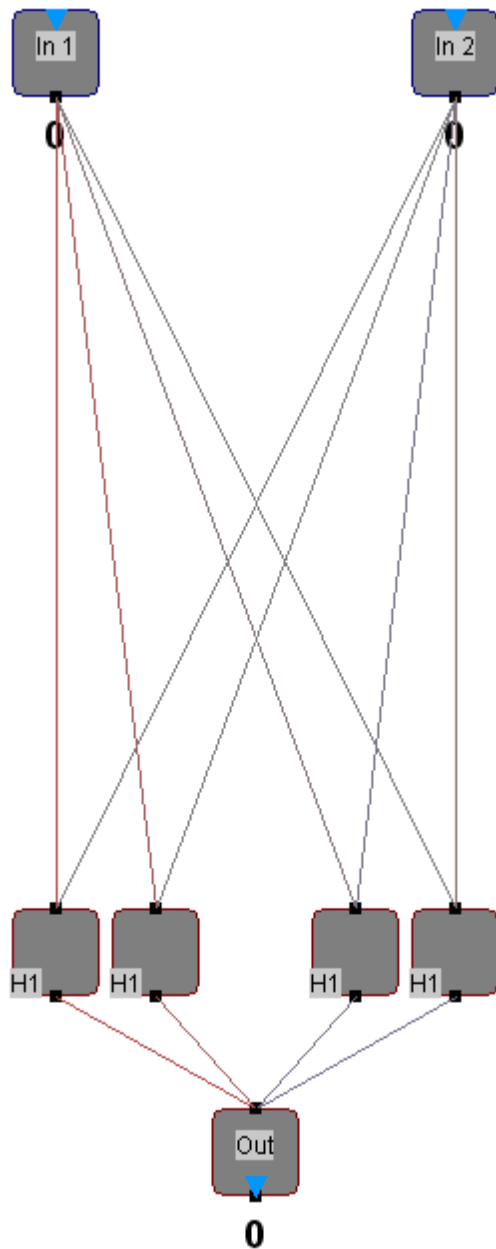
Step function response of a decay neuron.

Higher values for the Activation Sustain Factor cause more delay in the response, i.e. the neuron shows more inertia while smaller values make the neuron to follow the input signal more agile.

In order to set up decay neurons and their corresponding input links properly MemBrain provides a function that allows to create decay neurons automatically for all currently selected neurons.

The following example illustrates this. Note that for the screen shots the layer info display has been activated in the <View> menu and a net analysis has been enforced by selecting <Net><Analyse Net>. Also the net has been randomized.

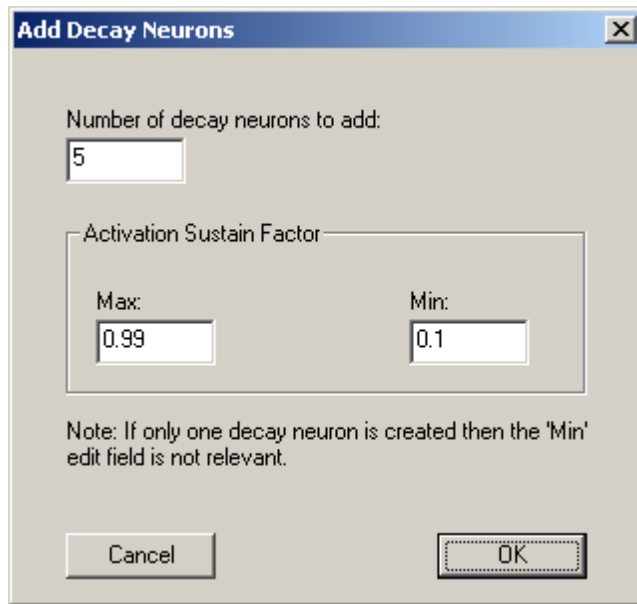
In the picture below a simple time invariant net with two inputs and one output is shown.



Simple time invariant net

Assume that this net shall be trained with patterns that represent a time series, i.e. the patterns in the corresponding lesson are ordered and the net shall learn the time series dependencies behind the patterns. In general, time invariant nets like the one shown above cannot accomplish such a task because these nets don't have any components which preserve information from the last calculation step (Think Step) into the next calculation step. Decay neurons are one possibility to incorporate the internal 'history' of the net into the next calculation step.

In order to add decay neurons to the inputs of this net select both input neurons and select the menu command **<Insert><Decay Neurons...>** or right click on one of the neurons and select the same command from the context menu. The following dialog will appear.



The dialog box is titled "Add Decay Neurons" and has a close button (X) in the top right corner. It contains a label "Number of decay neurons to add:" followed by a text input field containing the number "5". Below this is a section titled "Activation Sustain Factor" which contains two sub-sections: "Max:" with a text input field containing "0.99", and "Min:" with a text input field containing "0.1". At the bottom of the dialog, there is a note: "Note: If only one decay neuron is created then the 'Min' edit field is not relevant." and two buttons: "Cancel" and "OK".

Number of decay neurons to add:

5

Activation Sustain Factor

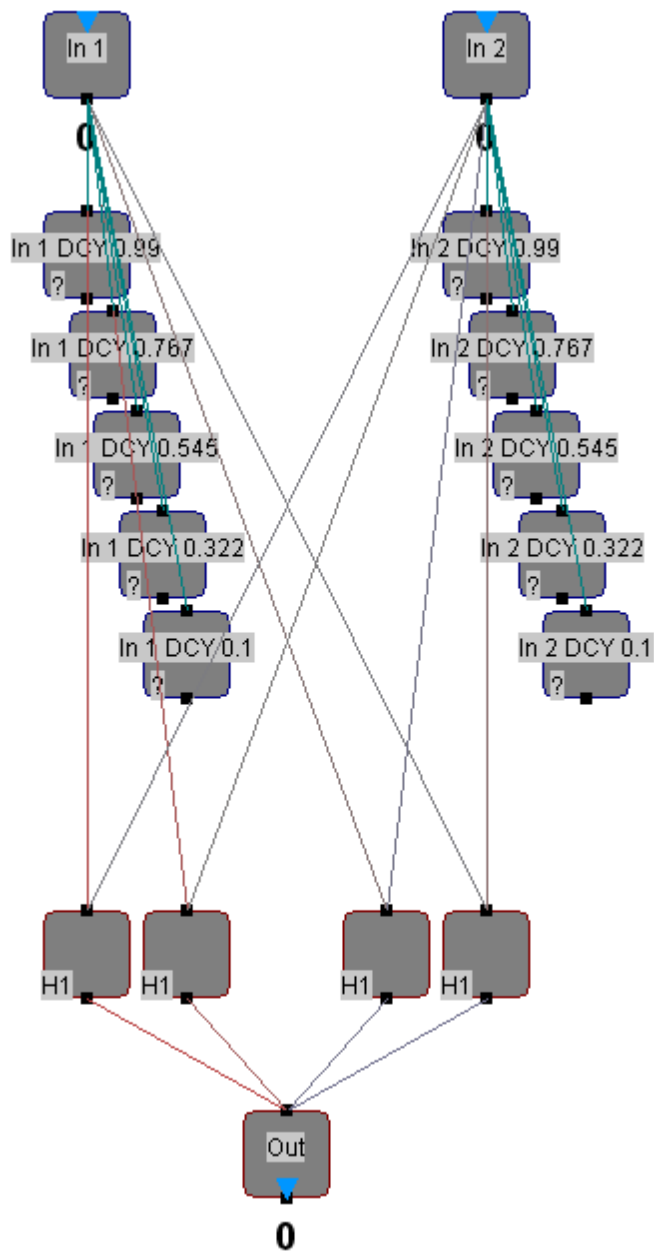
Max: 0.99 Min: 0.1

Note: If only one decay neuron is created then the 'Min' edit field is not relevant.

Cancel OK

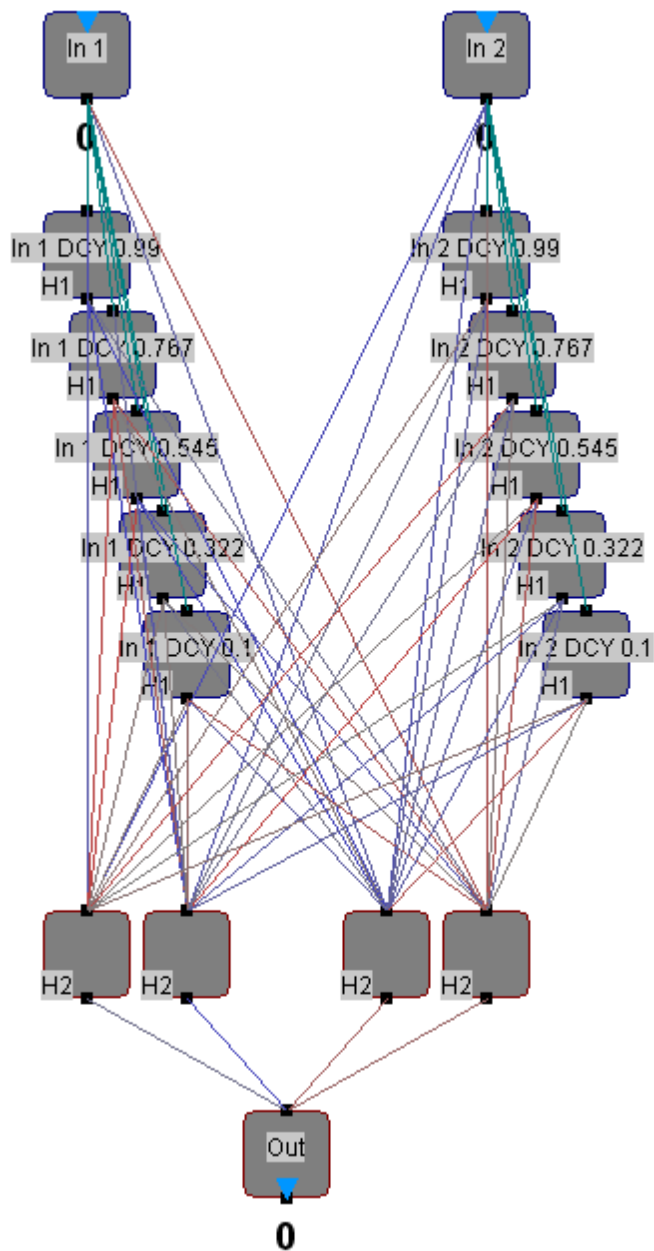
You can specify the number of decay neurons to be added to every selected neuron and the range of activation sustain factors that shall be used for the decay neurons.

Click on <OK>. The net now looks as following.



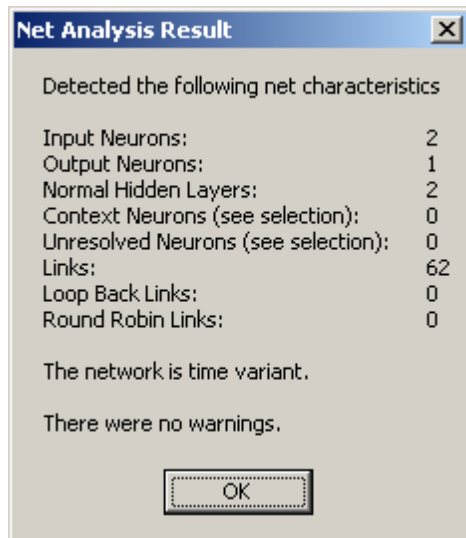
Note that 5 decay neurons with corresponding links have been added to each input neuron. The decay neurons are already properly configured and the links have been assigned the correct weights. Also the link weights and the activation thresholds of the neurons (which are set to 0) have been locked to prevent a teacher or the randomize function from changing them.

You can now connect the outputs of the decay neurons to the following layer(s) of the net to make the net look like this:



All neurons in the layer H2 receive inputs from the input neurons directly as well as from the decay neurons. Each decay neuron represents the history of its corresponding input neuron with more or less inertia depending on its activation sustain factor.

Now we have created a time variant network. You can verify this by selecting <Net><Analyse Net>:



Note the comment in the dialog box stating that this network is time variant.

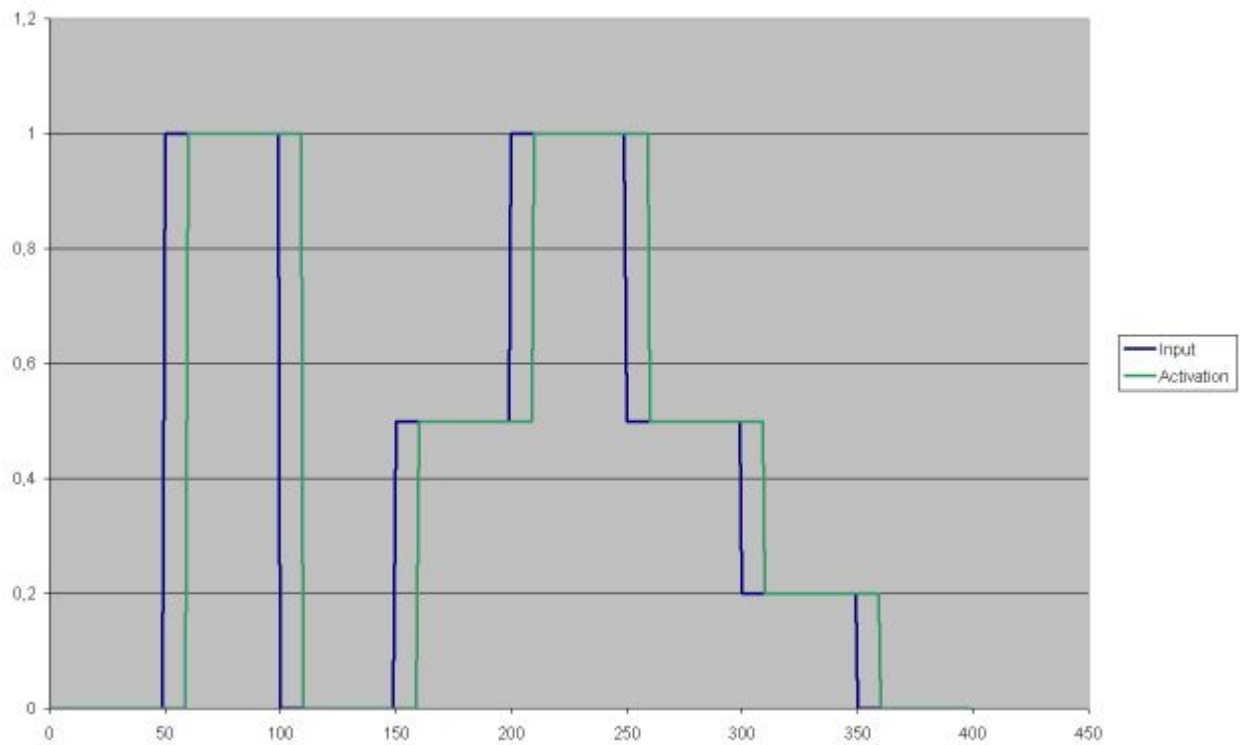
Adding Delay Neurons

Besides [Decay Neurons](#) another option to add time variant behaviour to your net is adding so called Delay Neurons. Both neuron types can also certainly be mixed in one and the same net.

A delay neuron is a neuron that is configured as a simple 'signal repeater'. However, the neuron is connected to its input source over a link with a logical length > 1 . The link with length > 1 builds up a delay line so that the signal at the output of the delay neuron is the same as the input signal but delayed in time.

Assume that the length of the input link to the delay neuron is n . Then the output signal will be delayed by $(n - 1)$ think steps.

The following chart shows the step function response of a delay neuron whose input link has a length of 11.



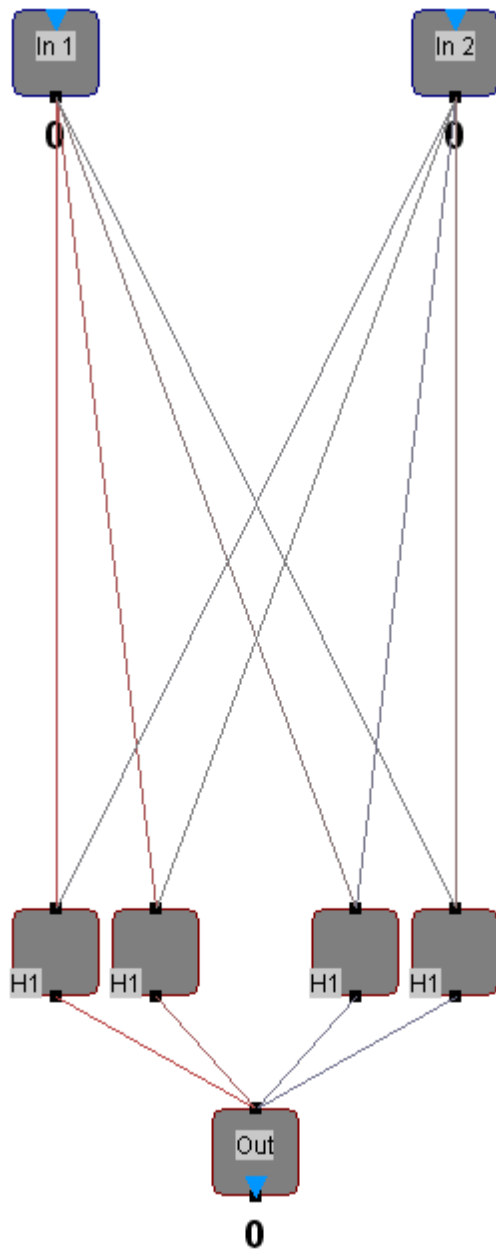
Step function response of a delay neuron.

Higher values for the link length cause more delay in the response.

In order to set up delay neurons and their corresponding input links properly MemBrain provides a function that allows to create delay neurons automatically for all currently selected neurons.

The following example illustrates this. Note that for the screen shots the layer info display has been activated in the <View> menu and a net analysis has been enforced by selecting <Net><Analyse Net>. Also the net has been randomized.

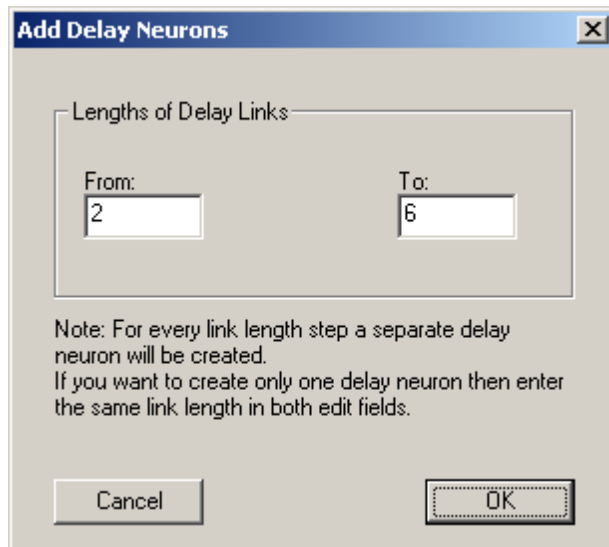
In the picture below a simple time invariant net with two inputs and one output is shown.



Simple time invariant net

Assume that this net shall be trained with patterns that represent a time series, i.e. the patterns in the corresponding lesson are ordered and the net shall learn the time series dependencies behind the patterns. In general, time invariant nets like the one shown above cannot accomplish such a task because these nets don't have any components which preserve information from the last calculation step (Think Step) into the next calculation step. Delay neurons are one possibility to incorporate the internal 'history' of the net into the next calculation step.

In order to add delay neurons to the inputs of this net select both input neurons and select the menu command **<Insert><Delay Neurons...>** or right click on one of the neurons and select the same command from the context menu. The following dialog will appear.



The dialog box is titled "Add Delay Neurons" and has a close button (X) in the top right corner. It contains a section titled "Lengths of Delay Links" with two input fields: "From:" containing the value "2" and "To:" containing the value "6". Below these fields is a note: "Note: For every link length step a separate delay neuron will be created. If you want to create only one delay neuron then enter the same link length in both edit fields." At the bottom are "Cancel" and "OK" buttons.

Add Delay Neurons

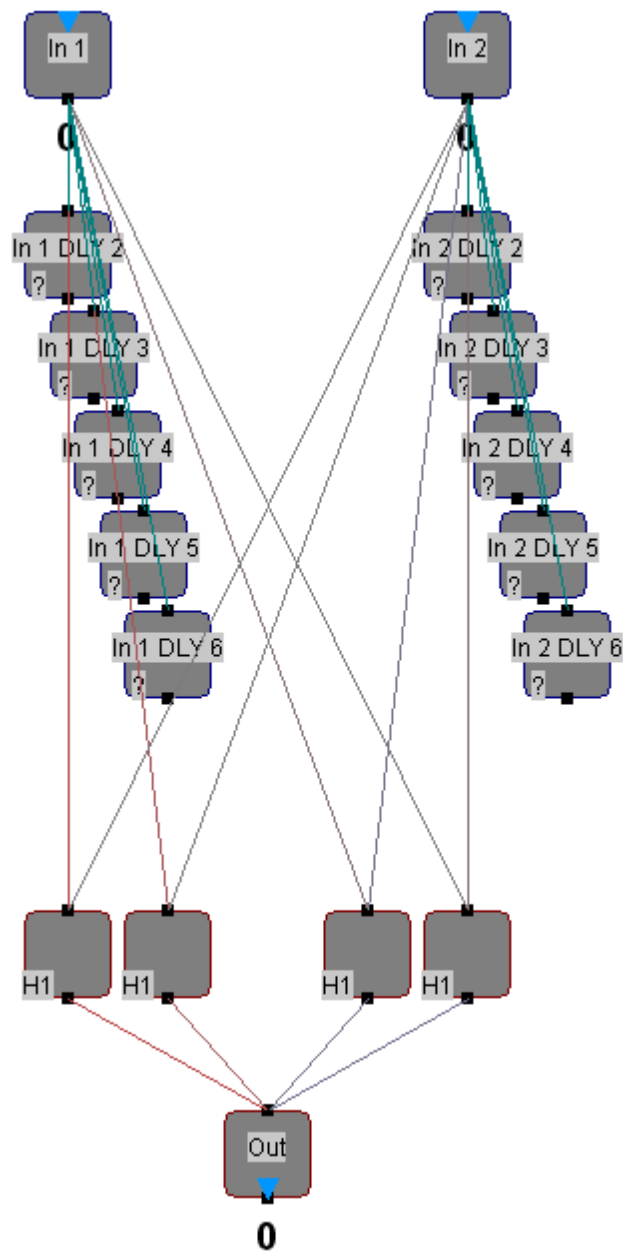
Lengths of Delay Links

From: To:

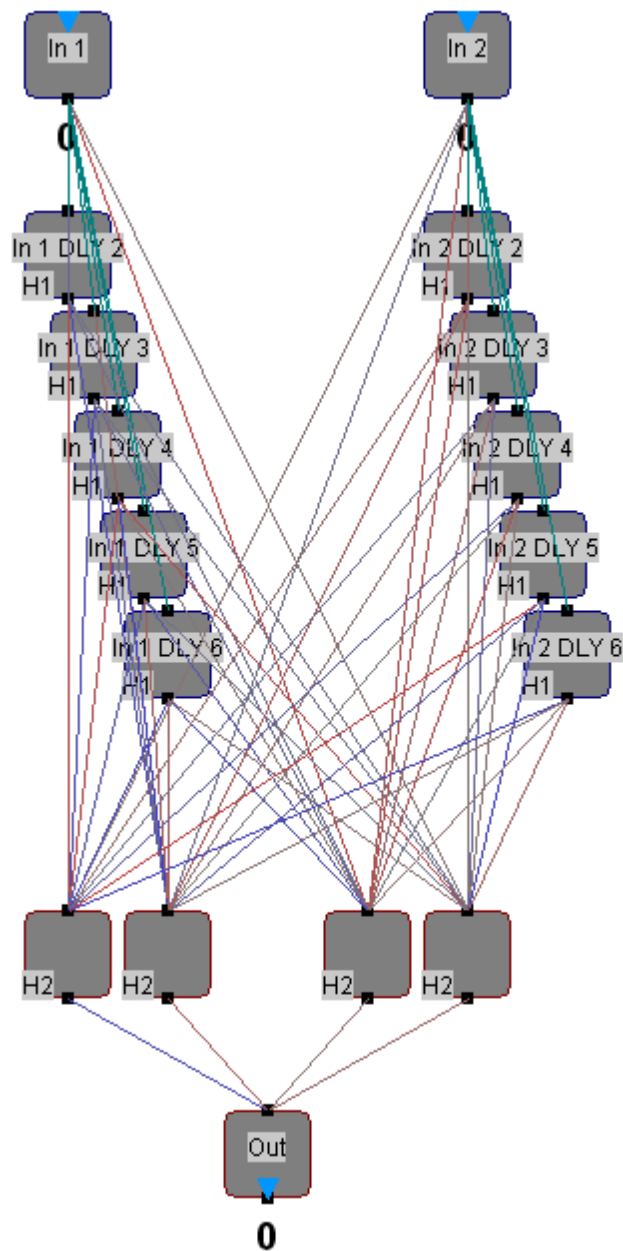
Note: For every link length step a separate delay neuron will be created.
If you want to create only one delay neuron then enter the same link length in both edit fields.

You can specify the range of link lengths to be used for every selected neuron. The difference between the 'To' link length and the 'From' link length determines how many delay neurons are created for every selected neuron: The number of created neurons always is 'To' - 'From' + 1.

Click on <OK>. The net now looks as following.

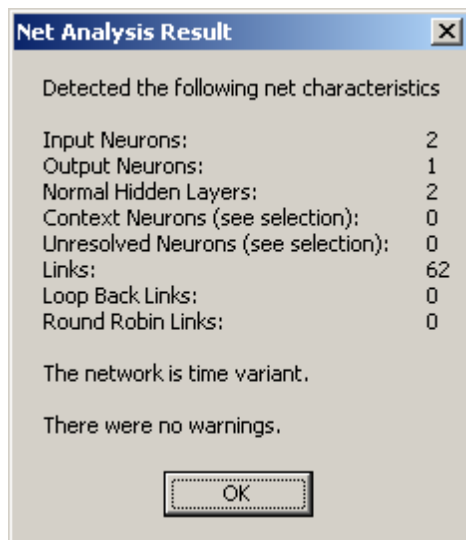


Note that 5 delay neurons with corresponding links have been added to each input neuron. The delay neurons are already properly configured and the links have been assigned the correct lengths. Also the link weights and the activation thresholds of the neurons (all set to 0) have been locked to prevent a teacher or the randomize function from changing them. You can now connect the outputs of the delay neurons to the following layer(s) of the net to make the net look like this:



All neurons in the layer H2 receive inputs from the input neurons directly as well as from the delay neurons. Each delay neuron represents the history of its corresponding input neuron with more or less delay depending on its input link's logical length.

Now we have created a time variant network. You can verify this by selecting <Net><Analyse Net>:



Note the comment in the dialog box stating that this network is time variant.

Adding Integrator Neurons

Integrator neurons are specially parametrized neurons which sum up signals over time. With each Think Step the input signal of the neuron is added to its activation.

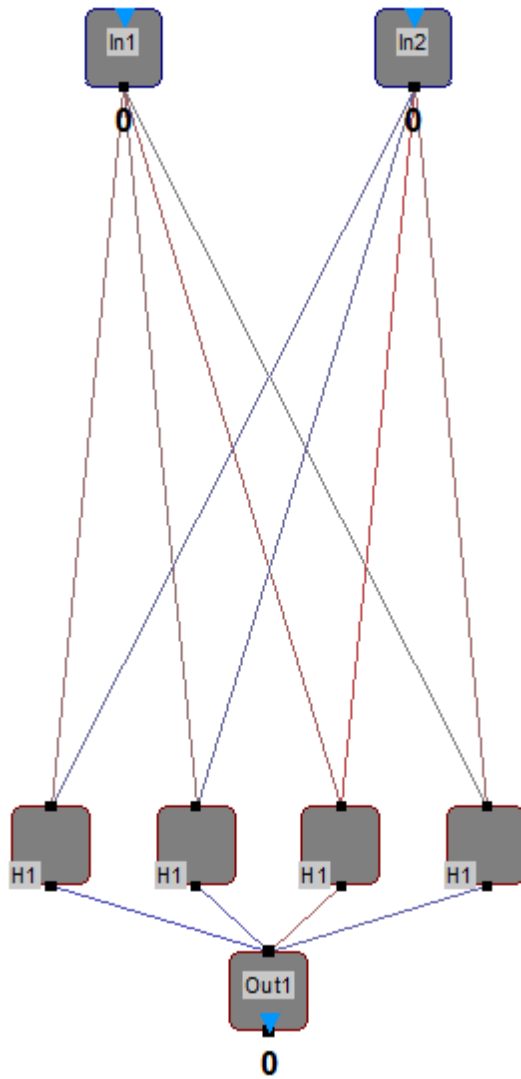
An integrator neuron is a hidden neuron whose 'Activation Sustain Factor' is set to 1. This means that this neuron incorporates its full current activation into the calculation of its next activation.

As a consequence, an integrator neuron can quickly reach its activation limits since its value else would exceed the internally allowed activation value range of $[-1 \dots 1]$. Thus, care must be taken to adjust the weight of the input link so that the incoming summands are small enough to allow for a sufficiently large number of Think Steps to be performed before the neuron would run into activation clipping.

In order to set up integrator neurons and their corresponding input links properly MemBrain provides a function that allows to create integrator neurons automatically for all currently selected neurons.

The following example illustrates this. Note that for the screen shots the layer info display has been activated in the <View> menu and a net analysis has been enforced by selecting <Net><Analyse Net>. Also the net has been randomized.

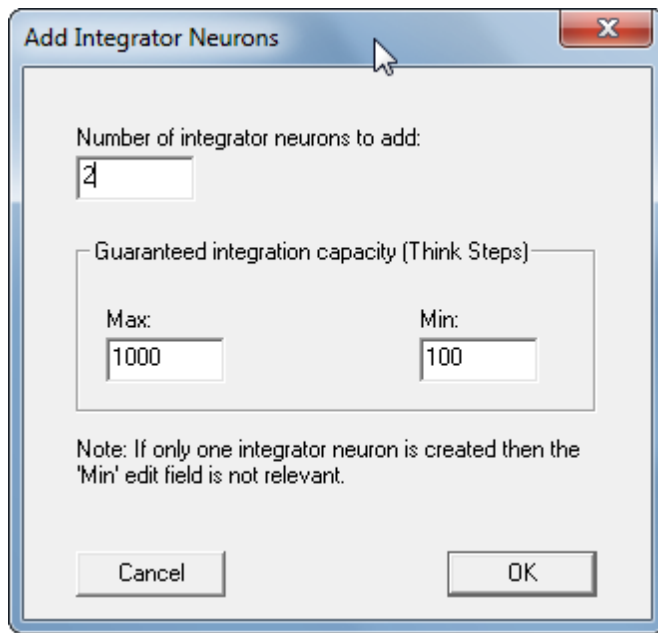
In the picture below a simple time invariant net with two inputs and one output is shown.



Simple time invariant net

Assume that this net shall be trained with patterns that represent a time series, i.e. the patterns in the corresponding lesson are ordered and the net shall learn the time series dependencies behind the patterns. In general, time invariant nets like the one shown above cannot accomplish such a task because these nets don't have any components which preserve information from the last calculation step (Think Step) into the next calculation step. Integrator neurons are one possibility to incorporate the internal 'history' of the net into the next calculation step.

In order to add integrator neurons to the inputs of this net select both input neurons and select the menu command **<Insert><Integrator Neurons...>** or right click on one of the neurons and select the same command from the context menu. The following dialog will appear. Note that the values entered in the input fields of the dialog have been adapted for this example.



The dialog box titled "Add Integrator Neurons" has a close button (X) in the top right corner. It contains a text label "Number of integrator neurons to add:" followed by a text input field containing the number "2". Below this is a section titled "Guaranteed integration capacity (Think Steps)" which contains two sub-sections: "Max:" with a text input field containing "1000", and "Min:" with a text input field containing "100". At the bottom, there is a note: "Note: If only one integrator neuron is created then the 'Min' edit field is not relevant." and two buttons: "Cancel" and "OK".

Number of integrator neurons to add:

2

Guaranteed integration capacity (Think Steps)

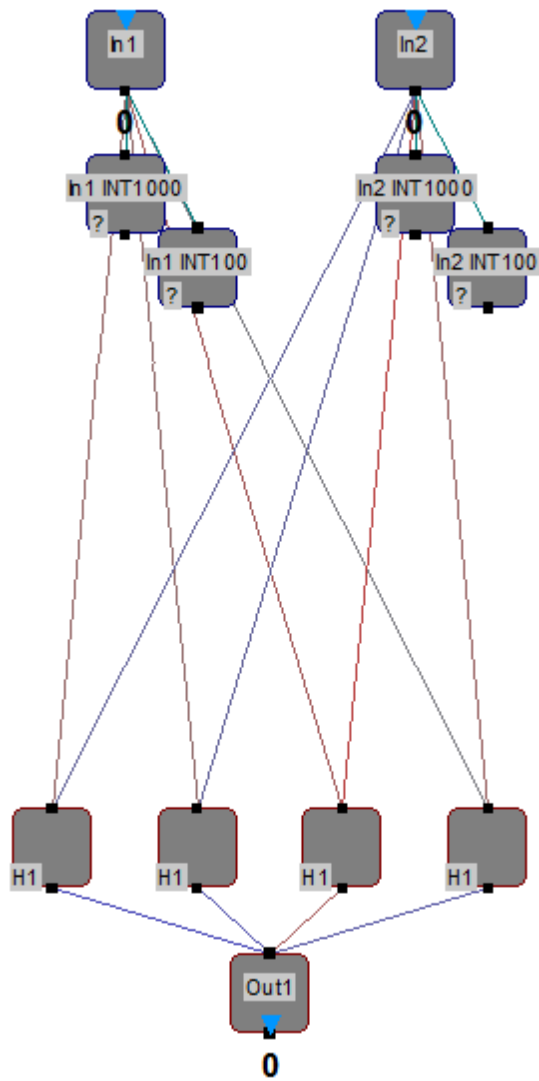
Max: 1000 Min: 100

Note: If only one integrator neuron is created then the 'Min' edit field is not relevant.

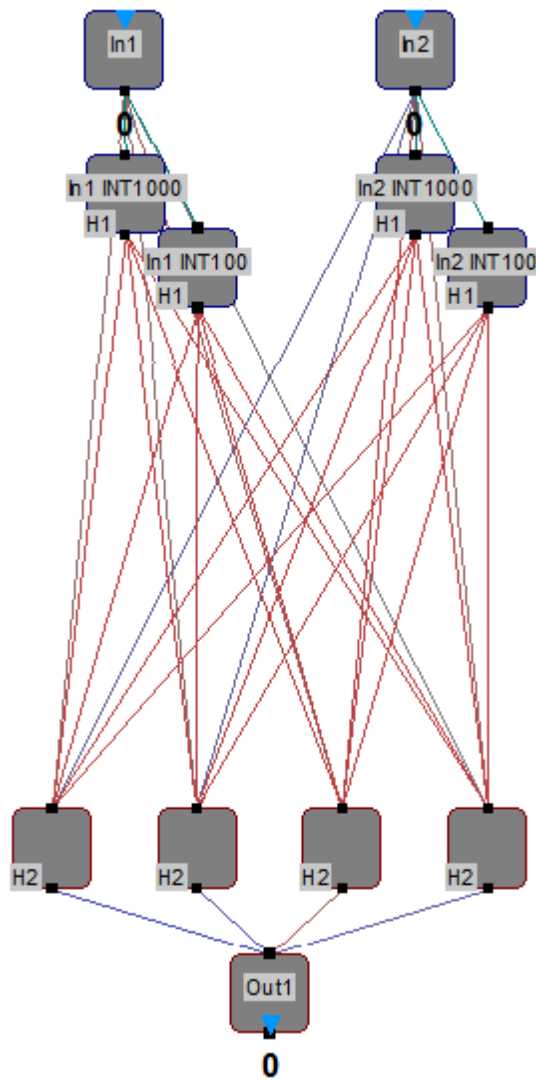
Cancel OK

You can specify the number of integrator neurons to be added to every selected neuron and the range of think step capacities that shall be used to dimension the input links of the integrator neurons.

Click on <OK>. The net now looks as following.

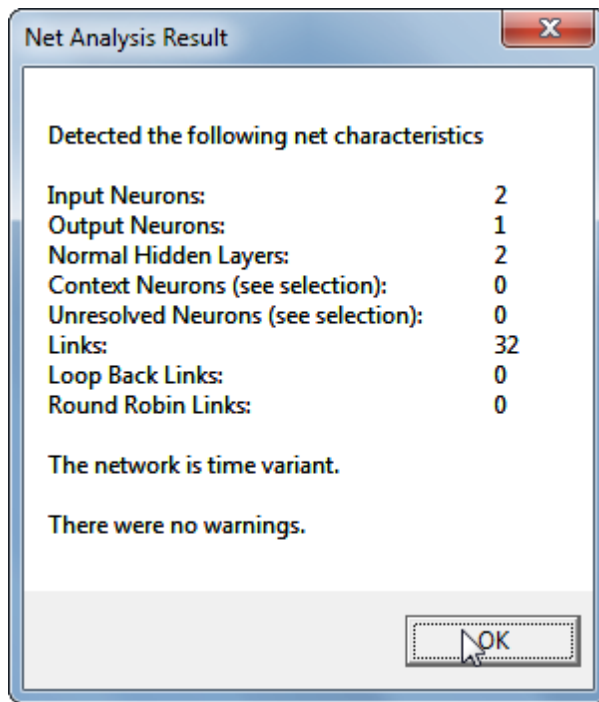


Note that 2 integrator neurons with corresponding links have been added to each input neuron. The integrator neurons are already properly configured and the links have been assigned the correct weights. Also the link weights and the activation thresholds of the neurons (which are set to 0) have been locked to prevent a teacher or the randomize function from changing them. You can now connect the outputs of the integrator neurons to the following layer(s) of the net to make the net look like this:



All neurons in the layer H2 receive inputs from the input neurons directly as well as from the integrator neurons. Each integration neuron represents the history of its corresponding input neuron in for of a sum over the last performed Think Steps.

Now we have created a time variant network. You can verify this by selecting <Net><Analyse Net>:



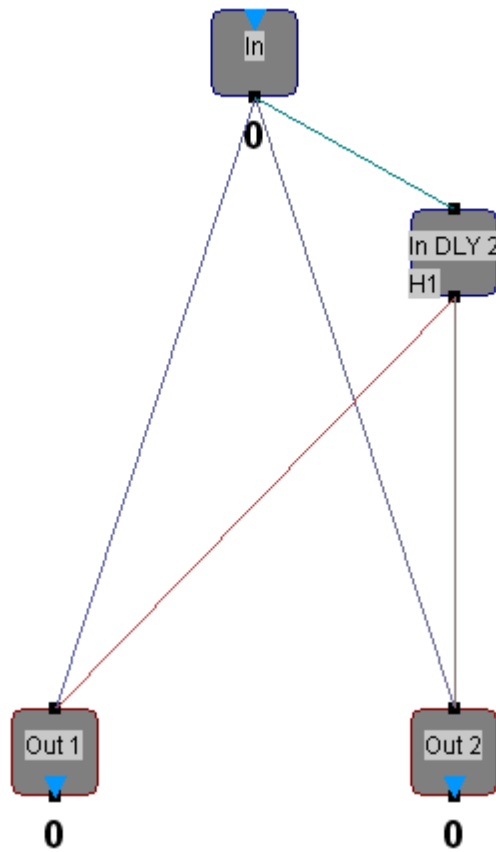
Note the comment in the dialog box stating that this network is time variant.

Add Differential Neuron

Sometimes it can be beneficial so create the difference between two neuron activations internally in the net.

E.g. if in a time variant net a neuron shall represent the rate of the change of an input with respect to its last value, i.e. the differential of a signal then a differential neuron can be added that represents the difference between the actual signal and a [delay neuron](#) that is connected to the same signal.

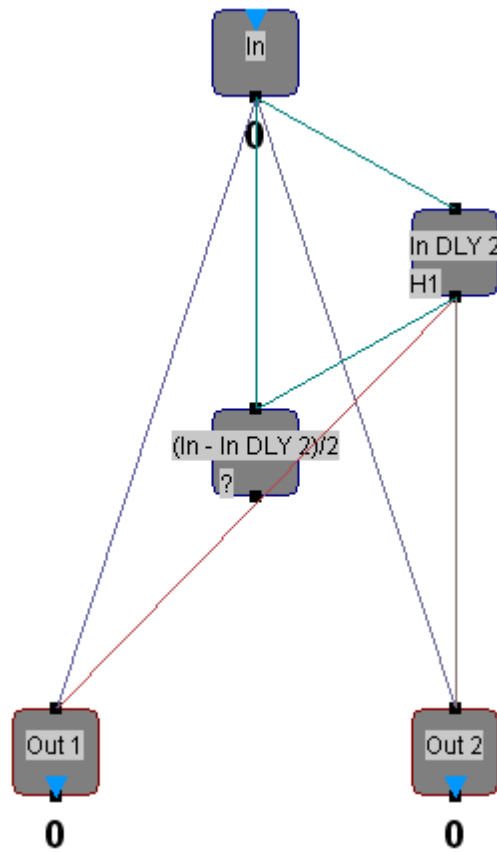
Consider the following net is given.



The input signal 'In' of that net already has been attached to a [delay neuron](#) with a link length of 2 which means that the delay neuron 'In DLY 2' represents the output signal of the input neuron 'In' one Think Step into the past.

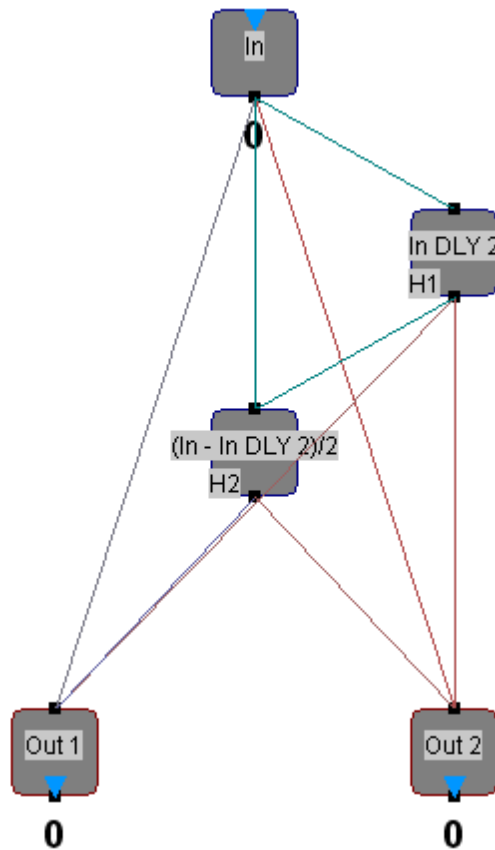
If you now want to add a differential neuron to the input neuron and its delay neuron you simply have to select both the neurons and then choose **<Insert><Differential Neuron>** or right click on one of the neurons and select the same command from the context menu.

The net now looks as follows.




Note that the new differential neuron has been assigned a name that reflects the difference between the neuron 'In' and the neuron 'In DLY 2', divided by two. Division by two is applied because the neuron shall be able to fully represent the difference between the two source neurons which can become between -2 and +2. The differential and the division by 2 are realized through the two newly created links which have fixed weights of +0.5 and -0.5, respectively.

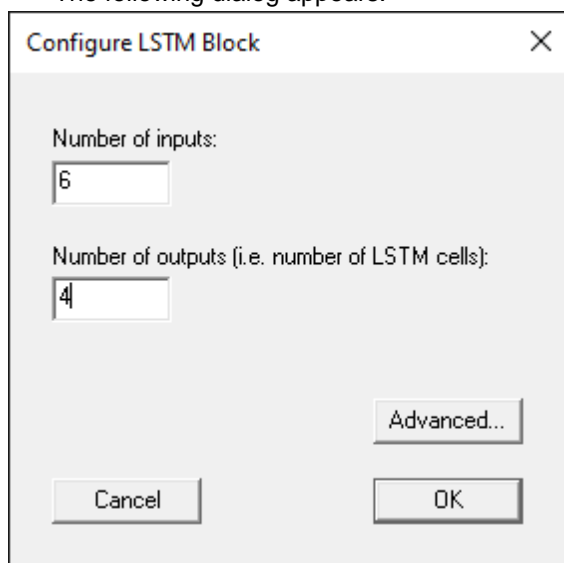
You can now connect the output of the new differential neuron as you like, e.g. connect it to the output neurons to get the following net.



Add LSTM Blocks

MemBrain provides the possibility to add LSTM (*Long Short Term Memory*) blocks:

- Select <Insert><LSTM Block...> or press the tool bar button 
- The following dialog appears:



The dialog box is titled "Configure LSTM Block" and has a close button (X) in the top right corner. It contains two input fields: "Number of inputs:" with the value "6" and "Number of outputs (i.e. number of LSTM cells):" with the value "4". Below these fields are three buttons: "Advanced..." (disabled), "Cancel", and "OK".

You can specify the number of inputs and outputs and have the possibility to configure advanced properties of the newly generated LSTM block. Note: The number of outputs is the number of internal LSTM cells of the

LSTM block. The inputs and outputs of the block (group) will be accessible via [proxy ports](#) after the generation.

The <Advanced...> button brings up the following options:

Configure LSTM Block: Advanced settings

Activation functions

F Gate: LOGISTIC O Gate: LOGISTIC I Gate: LOGISTIC

J Gate: TAN HYP C Transfer: TAN HYP

Peepholes to

☐ F Gate ☐ I Gate ☐ O Gate

Auxiliary features

☒ Incremental delay links per LSTM cell

☒ Incremental decay (i.e. sustain) function per LSTM cell

Max sustain: 0.1 Min sustain: 0.001

Note: If only one LSTM (decay) neuron is created then the 'Min' edit field is not relevant.

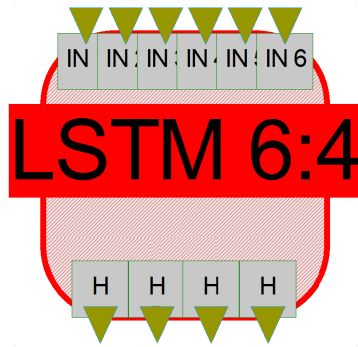
Cancel OK

Via this dialog it is possible to specify the used activation functions for the different gates inside the LSTM block. Additionally, it can be specified whether the LSTM block will feature so called peephole connections. Moreover, there are auxiliary, non-standard features that can be enabled:

1. Use incremental [delay lengths](#) to the context (time delay) neurons in each cell
2. Use incremental [decay](#) (sustain factor) configurations in the loopback links and the context (time delay) neurons in each cell

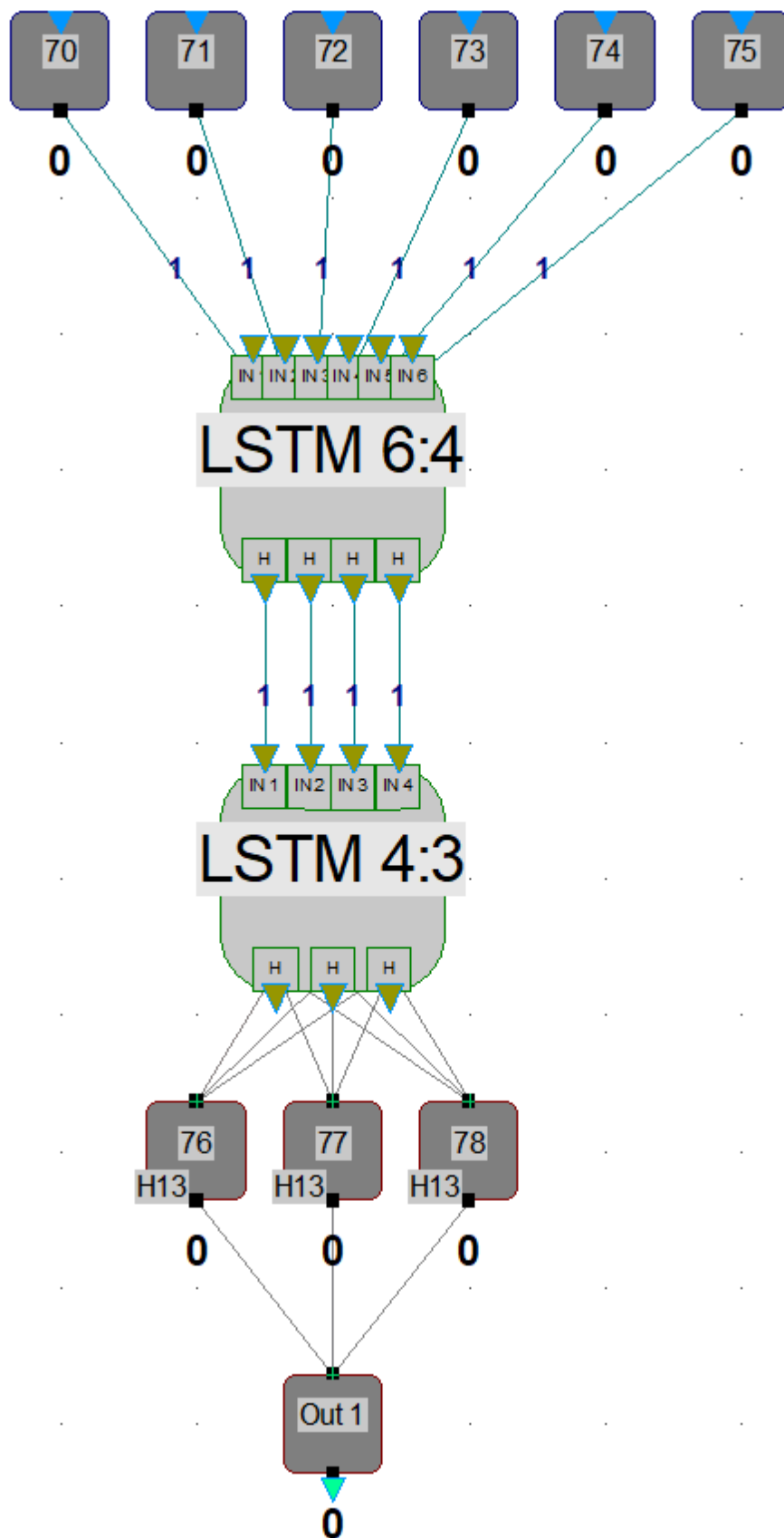
Note that the last entered settings will be saved persistently when MemBrain is exited. The default values which are also shown in the above picture can be restored as described [here](#) in case this should be required.

After clicking OK the LSTM block will be created as a new [group](#) of neurons and links which are configured and interconnected as required to form an LSTM block. The name of the new LSTM block/group is generated based on the specified number of inputs and outputs and can be changed as part of the normal [group properties](#) if desired.



Note that LSTM blocks are a specific type of a time variant network, i.e. their use makes the overall network time variant.

Typically, the inputs of an LSTM block are connected with fixed links with a weight of 1 to the input neurons of the overall net like shown in the following example. In this example, two LSTM networks are stacked and then followed by a fully connected hidden layer and an output layer (here: with dimension = 1).



Moving Neurons

In MemBrain neurons can easily be moved around in the drawing area:

- Select one or more neurons with the mouse (for details on how to select objects in MemBrain see [here](#))
- Hold the mouse button down over one of the selected neurons and move the whole selection to its new destination.
- Release the mouse button to place the neurons at their new position.

If the [Snap to Grid](#) option is activated then neurons will only move in steps of the current grid width. Else they will move without noticeable raster.

During moving neurons you can also [navigate](#) to regions of the drawing currently not visible without having to cancel the current move operation.

Note: You can also automatically align neurons in MemBrain. For details on how to do that click [here](#).

Aligning Neurons or Groups

Neurons and groups in MemBrain can be automatically aligned to each other with respect to their horizontal or vertical position. This is a useful feature when the [grid](#) is not activated because it is difficult then to align the objects manually.

Generally, there are two options of aligning objects in MemBrain:

1. Alignment to same horizontal or vertical position

This feature makes use of the [Extra Selection](#) capability of MemBrain:

- Select the target neuron or group to which a set of other neurons or groups shall be aligned.
- Apply [Extra Selection](#) to the target object.
- Select the neurons and groups that shall be aligned to the Extra Selected object.
- Click on the tool bar items



or

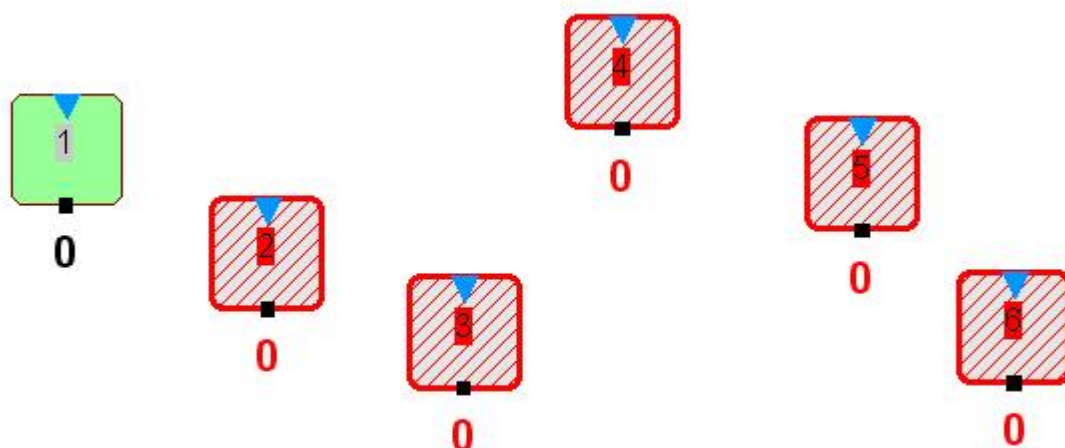


for horizontal respectively vertical alignment.

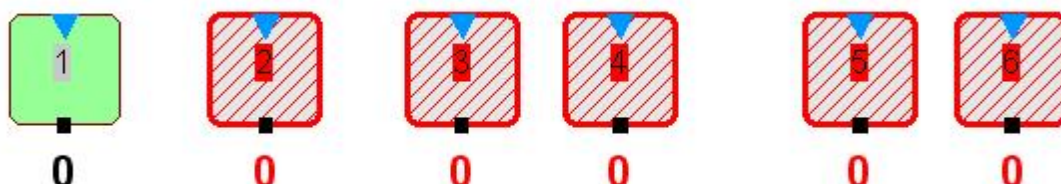
(Or choose <Edit><Align Horizontally> respectively <Edit><Align Vertically> from the main menu or from the context menu that appears when performing a right click on one of the selected neurons)

All selected neurons and groups will move horizontally/vertically to the same position as the Extra Selected object.

The following example shows the situation before and after performing the <Align Vertically> command.



Example situation before to execute the Vertical Alignment command



Example situation after execution of the Vertical Alignment command

1. Alignment to achieve equal horizontal or vertical spacing

If you want to have a set of neurons or groups being positioned with equal horizontal or vertical spacing just select the whole set of objects and press



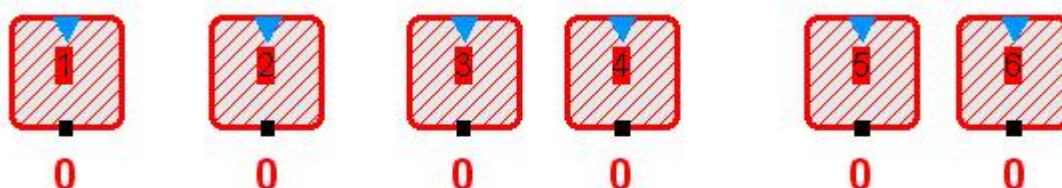
or



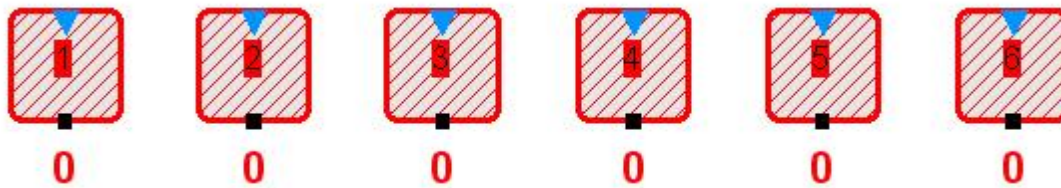
on the tool bar.

The same functionality can be achieved by selecting <Edit><Equal Spaces horizontally> respectively. <Edit><Equal Spaces vertically> from the main menu or from the context menu that appears when performing a right click on one of the selected objects.

The following example shows the situation before and after performing the <Equal Spaces horizontally> command.



Example situation before to execute the Equal Spaces horizontally command



Example situation after execution of the Equal Spaces horizontally command

Changing Neuron Properties

To edit the properties of one or more neurons select them with the mouse and do one of the following.

- Right click on one of the selected neurons and choose <Properties> from the context menu
- Select <Edit><Properties> from the main menu
- Double click on one of the selected neurons (also works with single neurons that are not yet selected).
- Hit <ENTER> on the keyboard.

Note: For more information on the various available methods of selecting objects on the screen see [here](#).

The following dialog will open.

Edit Object Properties [X]

Edit Neuron Properties | **Customize Activation Functions**

General

Name: ☒ Change

Type: ☒ Change

☐ Preferred Context Candidate

Input Function: ☒ Change

Activation Settings

Activation: ☒ Change

Activation Function: ☒ Change

Activation Threshold: ☒ Change

Activation Sustain Factor: ☒ Change

☐ Lock Act. Thres. for Teacher ☒ Allow Teacher to Connect Output

Output Settings

Output Fire Level: ☒ Change

Output Recovery Time: ☒ Change

Lower Fire Threshold: ☒ Change

Upper Fire Threshold: ☒ Change

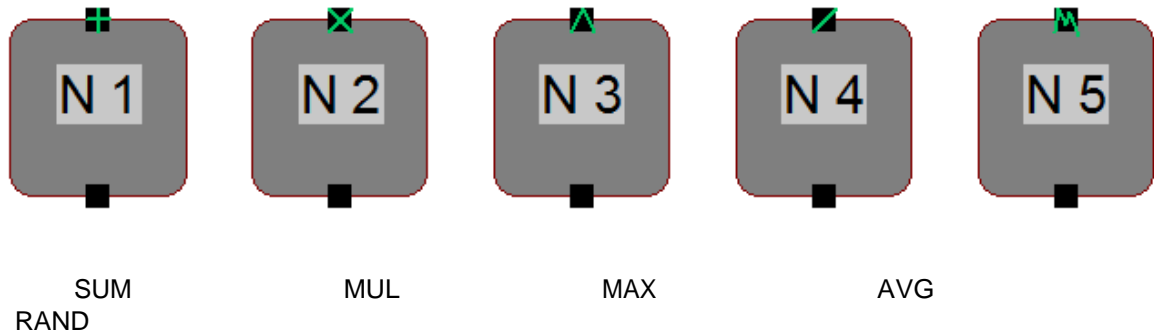
If you need more information about how these parameters influence the neuron's behaviour then please refer the section about [MemBrain's Neuron Model](#).

The fields and their meanings:

- Name:
Free selectable name of the neuron. The name will be displayed in the center of the neuron. Note that MemBrain also includes a [Neuron Auto Naming](#) feature that can be used if a larger group of neurons shall be re-named using a numbering scheme.
- Type:
One of the following
 - INPUT
 - HIDDEN
 - OUTPUT
- Preferred ContextCandidate
Only available for hidden neurons. If this box is checked then the neuron is preferably selected over other candidates as context neuron during the automatic [net analysis](#).
- Input Function:
One of the following

- SUM (Sum of all input signals)
- MUL (Product of all input signals)
- MAX (Maximum of all input signals)
- AVG (Average of all input signals)
- RAND (Random selection of one of the input signals)

The input function is visualized for each neuron in MemBrain by a symbol placed on its input connector. The following picture shows all input functions from above:



- Activation:
This is the current activation of the neuron. The activation together with the parameters in the dialog's 'Output Settings' section determine the signal that is output by the neuron.
Note: You can also set the activation of a neuron by the [Quick Activation](#) feature.
- Activation Threshold:
The Activation Threshold of the neuron is used for calculating its activation out of the previous activation, the sum of the input signals and the neuron's activation function. The activation function is a parameter that is changed during the teaching process if not explicitly locked (see further down this page on information about locking the threshold).
- Activation Function:
One of the following
 - LOGISTIC
 - IDENTICAL
 - IDENT. 0 TO 1
 - TAN HYP
 - BINARY
 - MIN EUCLID DIST
 - IDENT. -1 TO 1
 - RELU
 - SOFTPLUS
 - BINARY DIFF.

The activation function determines how the activation calculates out of the sum of the inputs and the Activation Threshold. See [here](#) for details on the available activation functions in MemBrain.

- Activation Sustain Factor:
The new activation of a neuron always calculates out of the current input signals and a certain fracture of the current activation. The Activation Sustain Factor determines which portion of the current activation flows into the calculation of the new activation.
- Lock Act. Thres for Teacher:
When this box is checked then the Activation Threshold of the neuron will neither be changed during teaching the net nor by the [Randomize Net](#) function. For more information about teaching a net click [here](#).

- Allow Teacher to Connect Output

If this option is deactivated then a teacher (learning algorithm) is not allowed to connect new links to the output of this neuron. This only has effects if the currently active teacher implements adding of new links, like the 'Cascade

Correlation' Teacher for example. If you are not sure then the best option is to leave the option activated since the teacher normally knows best which neurons to connect to.

- Output Fire Level:

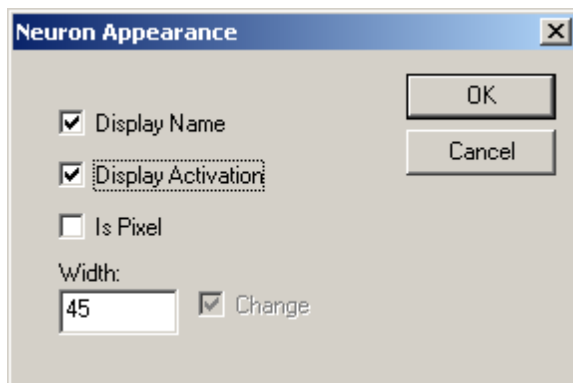
One of the following

- 1
- Activation

The Output Fire Level defines the output level of the neuron when firing. It is either a fixed value of 1 or the current (normalized) activation of the neuron, depending on the chosen setting.

- Appearance...:

If you click on that button you can edit the appearance of the neuron(s) via the following dialog.



- Display Name:

If this option is checked then the name of the neuron is shown on the neuron

- Display Activation:

When this box is checked the neuron's activation is printed on the screen below the neuron.

- Is Pixel:

If this property is set for a neuron it is displayed as a sharp edged rectangle that changes its color from black (activation = 0) over grey to white (activation = 1). If such neurons are placed close together on the screen they appear as a monochrome bitmap.

- Normalization...:

If you click on that button you can edit the normalization settings of the neuron(s) via the following dialog.

Enter the lower and the upper limit used when setting/displaying the activation of the neuron. The range given by these values will be normalized to the range defined by the activation function of the neuron.

Normalization is only effective to input and output neurons.

☐ Use Normalization

Normalization Range

Upper Limit: ☒ Change

Lower Limit: ☒ Change

You can also specify an activation value that will be used to detect if a certain output value of a certain Pattern in a Lesson shall be ignored during teaching.

This can be useful if correct output data is not available for some of the Patterns in a Lesson for example. If you set the value of these output data to the ignore value specified below then these output data will be ignored during teaching.

This setting is only effective to output neurons.

☐ Use Activation Ignore Value during Teaching:

Activation Value to be Ignored during Teaching

Value: ☒ Change

OK Cancel

Normalization means that you can specify a user specific activation value range for every input or output neuron of your net. This range will then be automatically converted internally by MemBrain to or from the real activation function range of the corresponding neuron as appropriate. Normalization is controlled by the following settings on this dialog:

- Use Normalization:
The neuron uses normalization only if this box is checked
- Lower and Upper Limit:
The range of input/output values that is normalized to the internal range of the activation function of the neuron.
- Use Activation Ignore Value during Teaching:
This setting allows you to optionally specify a special activation value for every output neuron of your net that serves as a place holder for data that shall be ignored during teaching.
Ignoring data during teaching may be useful if your data contains incomplete patterns, i.e. not all output data may be known for all of your patterns. In this case you can set the corresponding output data items of the patterns to this special value so that they will be ignored during the teaching process.

More information on Normalization of the activation function ranges of neurons is also available [here, together with the Normalization Wizard](#), that can assist you in setting the normalization ranges of your neurons.

The check boxes 'Change' are only functional if the properties of more than one neuron are edited. They are

- Grayed and checked if the corresponding setting is identical for all selected neurons (same as if only one neuron is selected).
- White and unchecked if the corresponding setting is different with the selected neurons. In this case the setting itself is grayed. If the box is checked with the mouse the setting gets editable and the new value will be applied to all selected neurons when the dialog is closed using <OK>. The box can be unchecked again. Then the corresponding settings are not changed when closing the dialog.

For details on the other tabs of the dialog refer to either [Changing Link Properties](#) or [Activation Functions](#).

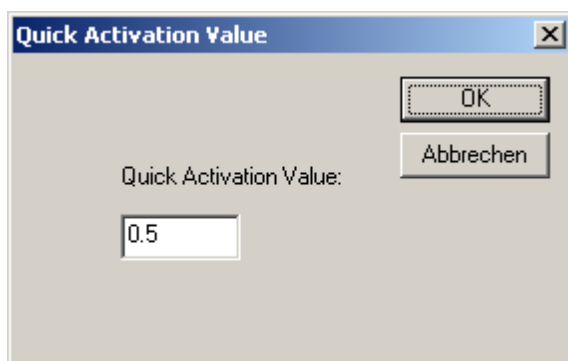
Click [here](#) for more information on how to add new neurons to the neural net.

Quick Activation

The activation values -1, 0 and 1 as well as a user definable fourth value can be assigned to neurons very quickly and easily. This is a useful feature when you want to quickly check the answer of a net to some test data created on the fly. Or maybe you want to draw a binary picture on an array of input neurons. You could then use the [Paint Brush Selection](#) method to select the neurons of interest with the mouse and afterwards activate the selected neurons by using quick activation.

Here's how to do so:

- Select the target neurons (see [here](#) for details on how to select items in MemBrain).
Then:
- Press <0> on the keyboard to apply an activation of 0
- Press <1> on the keyboard to apply an activation of 1
- Press <2> on the keyboard to apply an activation of -1
- Press <Strg + 1> on the keyboard or select <Extras><Apply Quick Activation> to apply a configurable activation value. The value can be configured by selecting <Extras><Quick Activation Value...> from the main menu which will open the following dialog.

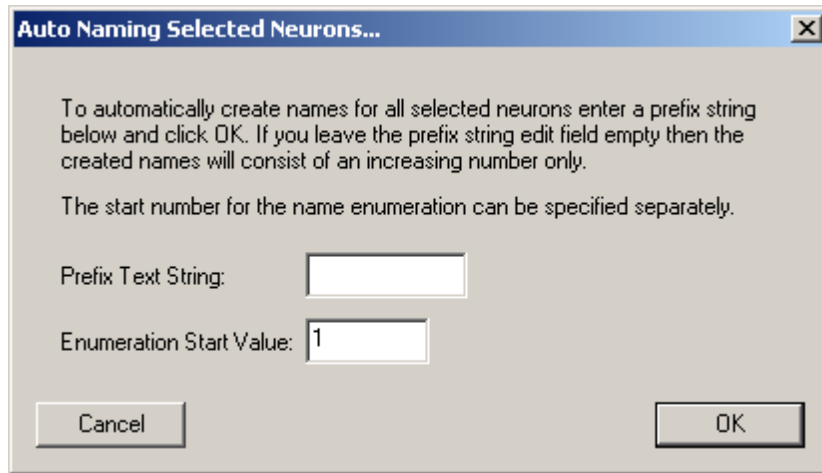


Neuron Auto Naming

MemBrain provides the possibility to automatically create and assign enumerated names to a couple of selected neurons:

- 1.) Select the neurons that shall be assigned an automatically generated name.
- 2.) Select the menu option <Edit><Neuron Auto Naming...>

The following dialog will open:



You can enter an optional prefix text and the start value for the auto name generation.

If you click on OK then all selected neurons will be automatically assigned an auto generated name according to the settings in the dialog.

Enumeration of the selected neurons will take place according to their location on the screen: The auto enumeration starts with the top left selected neuron and then goes down from top to bottom in rows from left to right.

Certainly, like any other changes on the net, the auto enumeration can be undone by the Undo/Redo functionality of MemBrain.

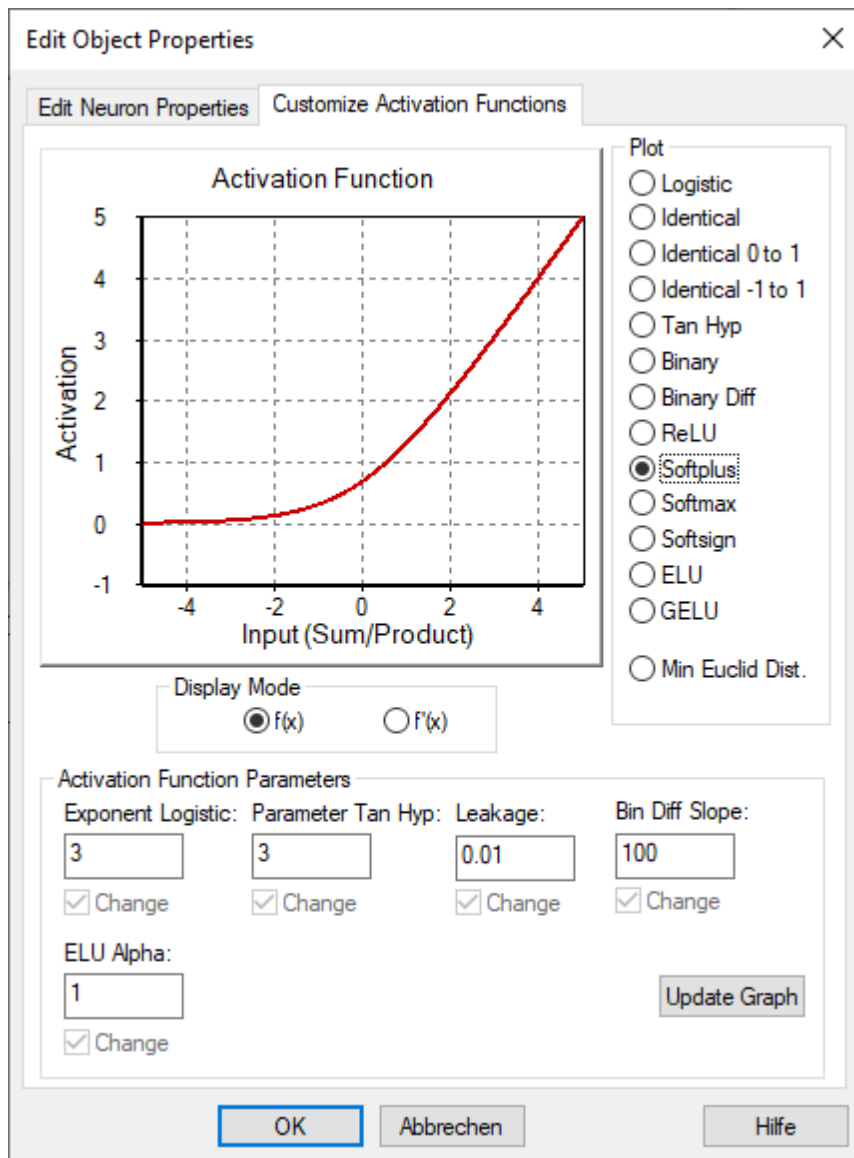
Activation Functions

MemBrain provides the following activation functions:

- LOGISTIC (i.e. Sigmoid)
- IDENTICAL (i.e. fully linear with slope = 1)
- IDENTICAL 0 TO 1 (slope = 1 between 0 and 1. Optional leakage slope outside this range)
- IDENTICAL -1 TO 1 (slope = 1 between -1 and 1. Optional leakage slope outside this range)
- TAN HYP (Tangens hyperbolicus)
- BINARY (either 0 or 1)
- BINARY DIFF (i.e. Differentiable Binary. Adjustable slope between 0 and 1. Optional leakage slope outside this range)
- RELU (Rectified Linear Unit. Slope = 1 between 0 and positive infinity. Else 0. Optional leakage slope below 0)
- SOFTPLUS (Softplus activation function). Always > 0. Increasing slope for values > 0. Approaches slope = 1.
- SOFTMAX (Softmax activation function). Exponential activation with additional postprocessing so that all SOFTMAX activations within the same layer sum up to 1.
- SOFTSIGN (Softsign activation function). Always between -1 and 1.
- MINIMUM EUCLIDEAN DISTANCE

Some of the activation functions can be parameterized individually for every neuron using the Properties dialog. For more information on editing neuron properties see [here](#).

The Properties dialog contains an extra tab for visualization and customization (if applicable) of the activation functions:



A graph can be displayed for every available activation function. Which one is being displayed can be selected on the right side of the graph area.

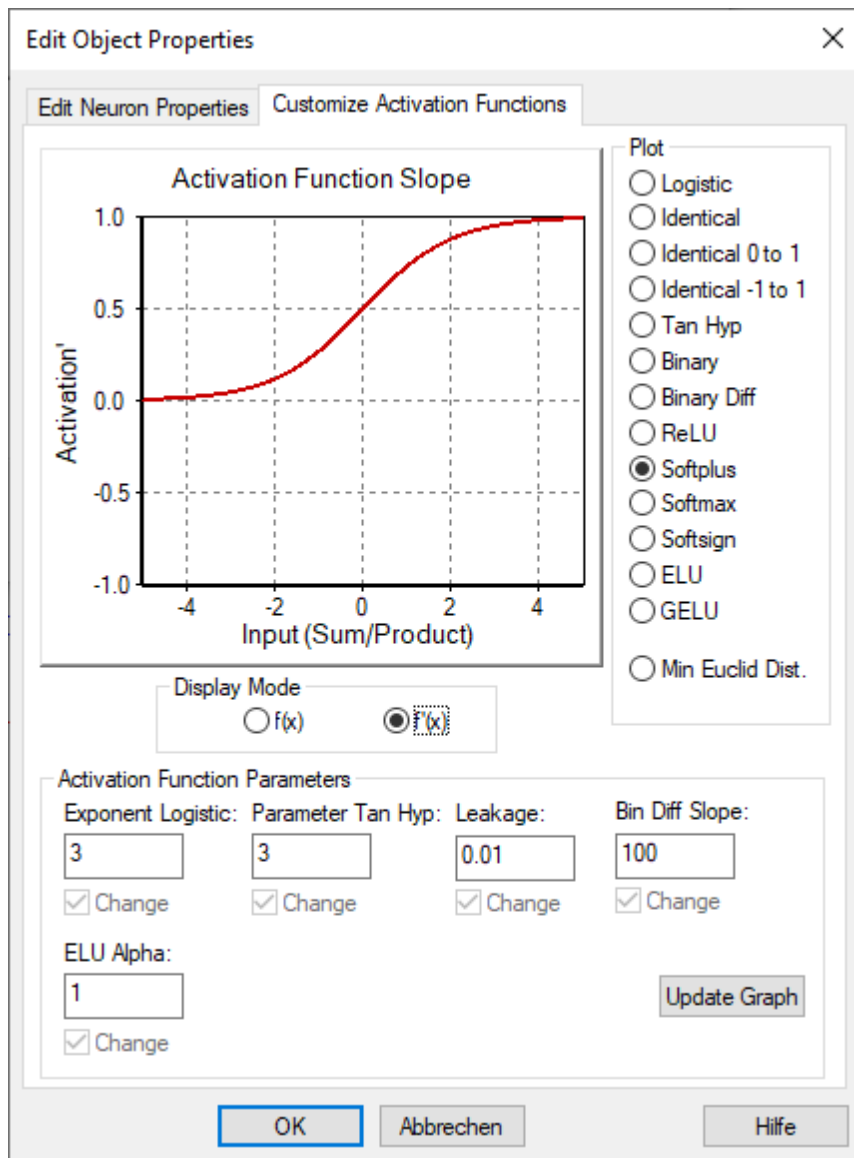
Note:

For plotting the graph an Activation Threshold of 0 is assumed. Also the current selection for the activation function being displayed does not influence the activation function selection made on the tab 'Edit neuron properties'.

For the Logistic function the parameter 'Exponent Logistic' and for the Hyperbolic Tangent function the Parameter 'Tan Hyp' can be adjusted. The parameter 'Bin Diff Slope' is only used for the activation function 'Binary Diff'. The 'Leakage' parameter is applicable for several activation functions, see list above. The button 'Update Graph' causes the graph to be refreshed after changing any of these values.

For the functionality of the check boxes named 'Change' see [here](#).

Moreover, the slope/derivative of the selected activation function can be displayed by selecting the radio button "f'(x)" below the diagram instead of "f(x)":



It should be mentioned that the activation function 'Minimum Euclidean Distance' is different from all other activation functions in principal: All other activation functions calculate the neuron's activation using the Input Sum or Input Product over all neuron inputs as a basis.

The 'Minimum Euclidean Distance' does not use the input sum or product but calculates the Euclidean distance between the input vector (i.e. the output vector of all neurons connected to the neuron's input) and the vector built by the weights of all input links of the neuron. This Euclidean distance is normalized to a length of 0..2 indicating how close these two vectors are together: An Euclidean distance of 0 means that the vectors are identical which will result in an activation value of 1. An Euclidean distance of 2 means that the differential vector between the two vectors has a length of 2 which means that one vector is the negative of the other. This will result in an activation value of 0.

The activation function 'Minimum Euclidean Distance' is used for the output neurons of so called Self Organizing Maps (SOMs) where similarity between input vector and weight vector is the criterion to determine the winner output neuron of the net, i.e. the neuron with the highest activation value. A derivative/slope is not defined for this function, thus the corresponding radio button is not functional in this case.

Neuron Model And Operation

All neurons in MemBrain are basically identical. Nevertheless they can behave totally different as a result of how their parameters are adjusted.

This chapter describes which mathematical model serves as the basis for neurons in MemBrain and explains the meaning of the different parameters which are part of every neuron's [properties](#).

When a new state of the neural net shall be calculated during simulation (this is called one 'Think Step' in MemBrain) then every neuron performs its own calculation based on the signals on its inputs, its current activation and its parameter settings. The result of this calculation determines the new activation of the neuron which in turn produces the signal that appears on the output of the neuron according to another set of adjustable neuron parameters.

The output signal of a neuron is then transferred over all connected output links to the input ports of the destination neurons to which the links route to. The signal is additionally multiplied by the weight of the corresponding link before it reaches the input port of the destination neuron.

To completely understand the behaviour of a neuron in MemBrain it is important to clearly distinct between a neuron's **Activation** and its **Output Signal**.

The Activation:

Let's take a look at the calculation of the activation first which is given by the following formula:

$$Act = Act * ActSustainFactor + ActFunc (Input - ActivationThreshold)$$

As we can see the new activation of a neuron calculates out of two parts:

- The current activation multiplied by a so called *Activation Sustain Factor*
- and
- The value of the *Activation Function* that receives the combined input value from all inputs of the neuron minus the so called *Activation Threshold* as argument (except for the minimum Euclidean distance activation function which calculates differently). The mentioned combined input usually is the sum of all inputs. It can, however, also be calculated by another input function. Which method for generating the input is used depends on the selected Input Function for the neuron which is part of the neuron's [properties](#).

Additionally, the resulting activation value may clipped to the range of [0..1] or [-1 .. 1] whatever range applies to the specific activation function.

To get more information about the available activation functions in MemBrain click [here](#).

For a first understanding lets assume the second part in the formula is 0. In MemBrain this is the case in two situations: Either the combined input value of all inputs together with the Activation Threshold causes the Activation Function to result in a value of 0 or the neuron has got no signals (i.e. no links) connected to its input port. *If a neuron has no signals connected to its input the whole right part of the formula is omitted. Note that this is not the same as an input signal of 0!*

In this case the new activation calculates out of the current activation and the Activation Sustain Factor only. This causes the activation of the neuron to decay with every calculation step until it reaches 0. The Activation Sustain Factor determines how quickly this will happen: A value of 0 causes the activation to immediately drop to 0 with the next calculation step. A value of 1 will cause the activation to stay unchanged forever. Any value in between these limits will produce an activation that exponentially decays to 0 over a certain number of calculation steps. (Theoretically the activation would never reach 0 with any value of the Activation Sustain Factor except for 0. However, the minimum representable value for a 64 bit floating point number (which is used by MemBrain) is approx. 1e-37. Such values are set to 0 by MemBrain 0 automatically.)

Together with the right part of the formula we get a good picture of how a neuron' activation behaves in the simulation:

Its inputs in conjunction with the **Activation Function** cause the activation to rise or drop depending on the signs of the inputs while the **Activation Sustain Factor** brings the activation back to 0 when no significant activation is caused by the inputs. The sensitivity of the neuron with respect to its inputs is determined by

the **Activation Threshold**.

The Output Signal:

After the activation of a neuron has been calculated according to the procedure above, its output signal is determined according to the following table: (An 'X' in the table means that the setting is not relevant, i.e. 'Don't Care')

| # | Activation | Output Fire Level | # of Calculation Steps Since Last Firing | Signal at Neuron Output |
|---|---|-------------------|--|-------------------------|
| 1 | Act <= Lower Fire Threshold | X | X | 0 |
| 2 | X | X | # < Output Recovery Time | 0 |
| 3 | Act >= Upper Fire Threshold | "1" | # >= Output Recovery Time | 1 |
| 4 | Act >= Upper Fire Threshold | "Activation" | # >= Output Recovery Time | Act |
| 5 | Lower Fire Threshold < Act < Upper Fire Threshold | "1" | # >= Output Recovery Time | 0 or 1 (*) |
| 6 | Lower Fire Threshold < Act < Upper Fire Threshold | "Activation" | # >= Output Recovery Time | 0 or Act (*) |

(*): The decision if the neuron is firing (output <> 0) or not (output = 0) is made on the basis of a firing probability that rises from 0 to 1 (= 100%) with the activation ranging from the Lower Fire Threshold to the Upper Fire Threshold.

Note that in the above table 'Act' always refers to the normalized activation of the neuron, i.e. is independent from user defined data ranges (the so called ['Normalization Settings'](#)).

As we can see the output of the neuron is determined by the variables

- Activation
- Number of Activation Calculation Steps since Last Firing of the neuron

in combination with the following parameters which are [properties](#) of every neuron:

- The Lower Fire Threshold
- The Upper Fire Threshold
- The Output Fire Level selection (either "1" or "Activation")
- The Output Recovery Time

The lines of the above table are explained in a little more detail now. The meaning of the term "fire" is that the output of the neuron has a value <> 0.

Line #1:

If the activation is smaller than or equal to the Lower Fire Threshold then a neuron won't fire in any case.

Line #2:

If the number of activation calculations since the last firing of the neuron is smaller than the Output Recovery Time the neuron won't fire in any case. The smallest allowed value for the Output Recovery Time is 1 which

generally enables a neuron to fire with every calculation step. (Use a setting of 1 if you want to disable the Output Recovery Time setting)

Line #3 and Line #4:

If the activation is bigger than the Upper Fire Threshold then the neuron will fire in any case as long as the Output Recovery Time setting allows for that (see Line #2).

Which value appears on the output when the neuron fires is determined by the Output Fire Level which has two possible settings: "1" or "Activation". When set to "1" then the neuron always fires with an output level of 1. In the latter case the neuron outputs its current activation when firing.

Line #5 and Line #6:

When the activation is bigger than the Lower Fire Threshold AND smaller than or equal to the Upper Fire Threshold then the neuron fires with a probability that continuously rises from 0 to 1 (= 100%) with increasing activation. Again, also the Output Recovery Time setting has to allow for that as described in Line #2.

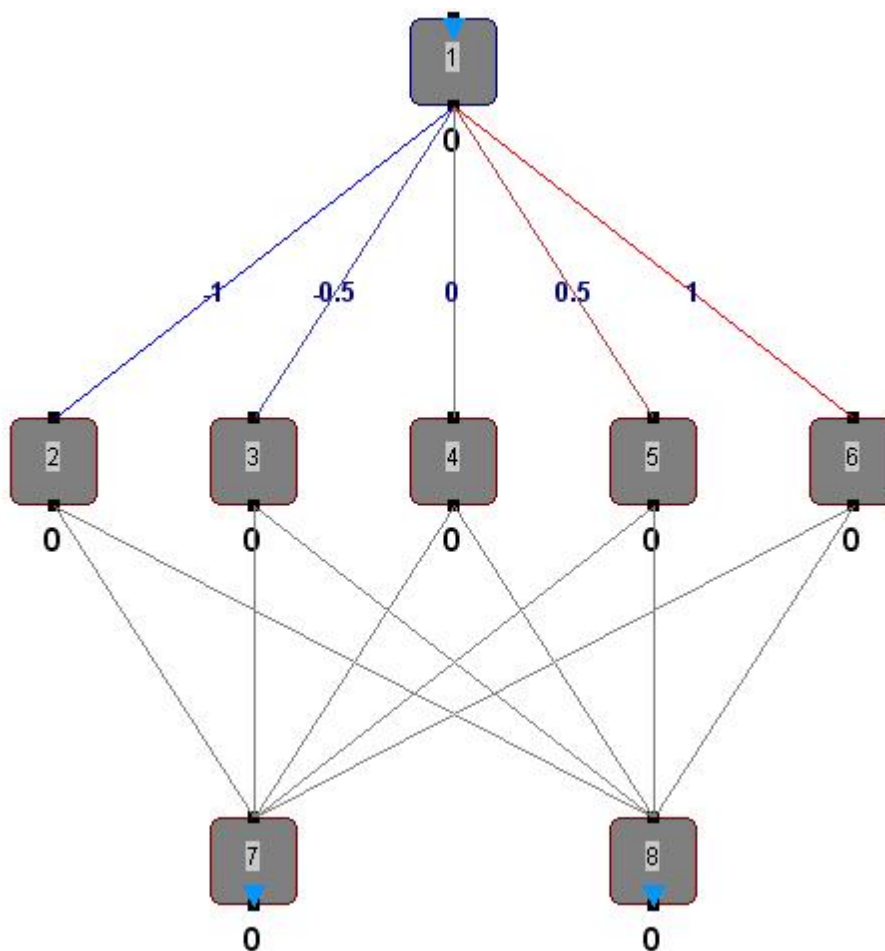
This brings in some random behaviour for a neuron when the Lower Fire Threshold and the Upper Fire Threshold are not identical. If you do not want to have this random behaviour then use the same setting for both of the thresholds. The output signal in case the neuron fires is again determined by the Output Fire Level setting as already described with Line #3 and Line #4.

Links in MemBrain

Neurons in MemBrain are connected to each other with links. This chapter gives general information about the links in MemBrain. For information on how to interconnect neurons with links, click [here](#). If you want to find out how to edit the properties of links in MemBrain see [here](#). If you want to learn more about the functionality of Links in MemBrain then please refer to the [Link Model](#) chapter.

A link is the weighted connection between a neuron's output connector at the bottom of the neuron and the input connector of another neuron or even the same neuron at its top.

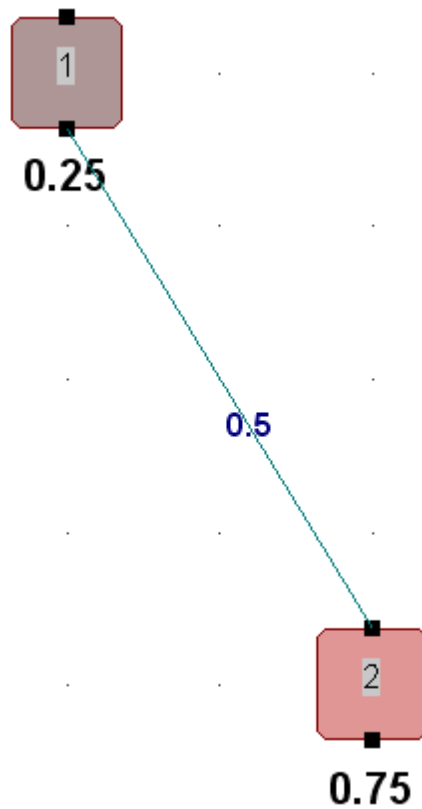
Links in MemBrain are displayed as thin lines between the connectors of the neurons and change their color according to their current weight beginning with blue at a weight of -1 or smaller, reaching gray when 0 and turning to red up to the point where the weight is 1 or bigger:



Note: Links are only displayed in MemBrain when the option <View><Show Links> from the main menu is activated. Also you can select if links are drawn in foreground or in background (behind the neurons). This is a useful option when many links are crossing neurons in the drawing. Choose <View><Draw Links in Foreground> to toggle between the two options.

As shown in the example the current weight of a link can be displayed together with the link. This is one of the [properties](#) that can be adjusted with every link.

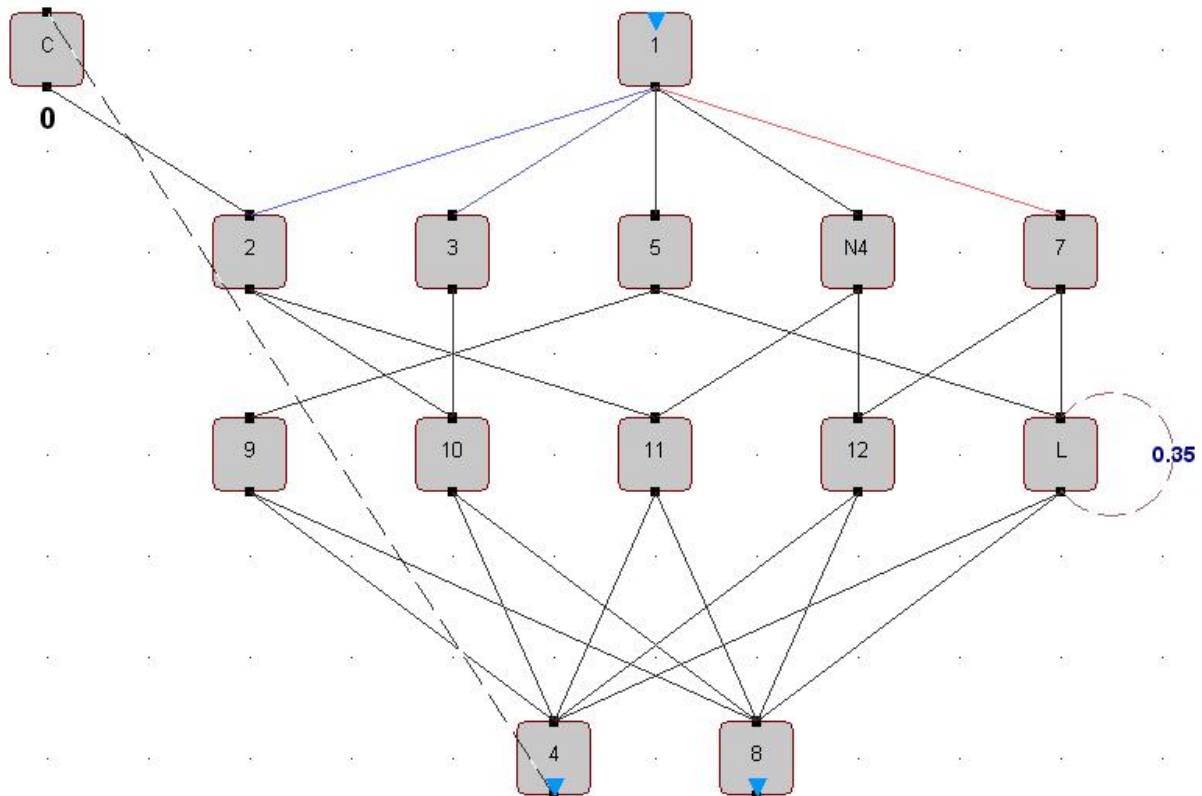
Another link [property](#) is the locking feature of its weight. If the lock flag of a link is set the weight of the link is not changed by the teaching process and by the Randomize Net command. A link with a locked weight is displayed magenta colored as shown below, independently from its actual weight value.



Link with a locked weight of 0.5

A link in a neural net can be logically forward or backward connecting. The links in the example given above are all forward links.

The following example net also contains backward (loop back) links which are displayed by dashed lines.



There are two loop back links in this net: One goes from the output neuron with name '4' backwards in the net's layer hierarchy to a so called 'context neuron' named 'C' in the example. The other one is actually a round robin link that connects the output of neuron 'L' with its own input.

For more information on the behaviour of a net that contains loopback links and what consequences that implies for teaching the net see chapter [Neural Networks](#) and the corresponding sub chapters.

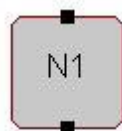
Note: When interconnecting two neurons, links that go upwards in the drawing area are assumed to be loopback links as long as no net analysis has been run. The net analysis will examine the net and determine which links in the net are real loop back links and which aren't. So the appearance of the links may change once the net analysis has been run. For more information on net analysis click [here](#).

Adding Links Between Neurons

This chapter describes how to add links between neurons.

Neurons in MemBrain always have their inputs on their upper and their output on their lower connector:

Input Connector



Output Connector

A link in MemBrain always must connect an output to an input. Generally spoken that is the only rule that applies to links between neurons.

In MemBrain there are many ways to connect neurons with links. Because of this the description of how to do so has been broken down into several sub chapters.

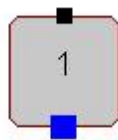
The possibilities are:

- [Manually create single links](#)
- Connect FROM the Extra Selection to the current selection
 - [Full linkage](#)
 - [Random linkage](#)
 - [1:1 linkage](#)
 - [Matrix based linkage](#)
- Connect TO the Extra Selection from the current selection
 - [Full linkage](#)
 - [Random linkage](#)
 - [1:1 linkage](#)
 - [Matrix based linkage](#)
- [Add output links to the current selection and connect them manually to the inputs of other neurons](#)
- [Add input links to the current selection and connect them manually to the outputs of other neurons](#)

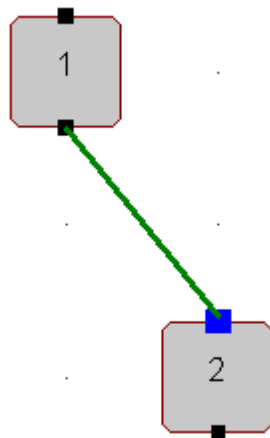
Single Connections

Note: For this functionality to be available the link display option of MemBrain must be enabled: Check that the option <View><Show Links> in MemBrain's main menu is active!

To manually connect two neurons simply hover the mouse over one of the connectors of a neuron (upper side connector is input, bottom side is output). The connector will automatically enlarge and turn to blue color indicating that a connection can be made:



Click with the left mouse button on the connector and a new link will be attached to it. Its other end is at the tip of the mouse cursor. Now hover the mouse over the target connector of another or even the same neuron. The connector will get highlighted if a connection is possible. Click again with the left mouse button:



The neurons are now connected with a single link.

Note: New links are always created with default properties. The default properties can be set by choosing <Edit><Default Properties...> from the main menu.

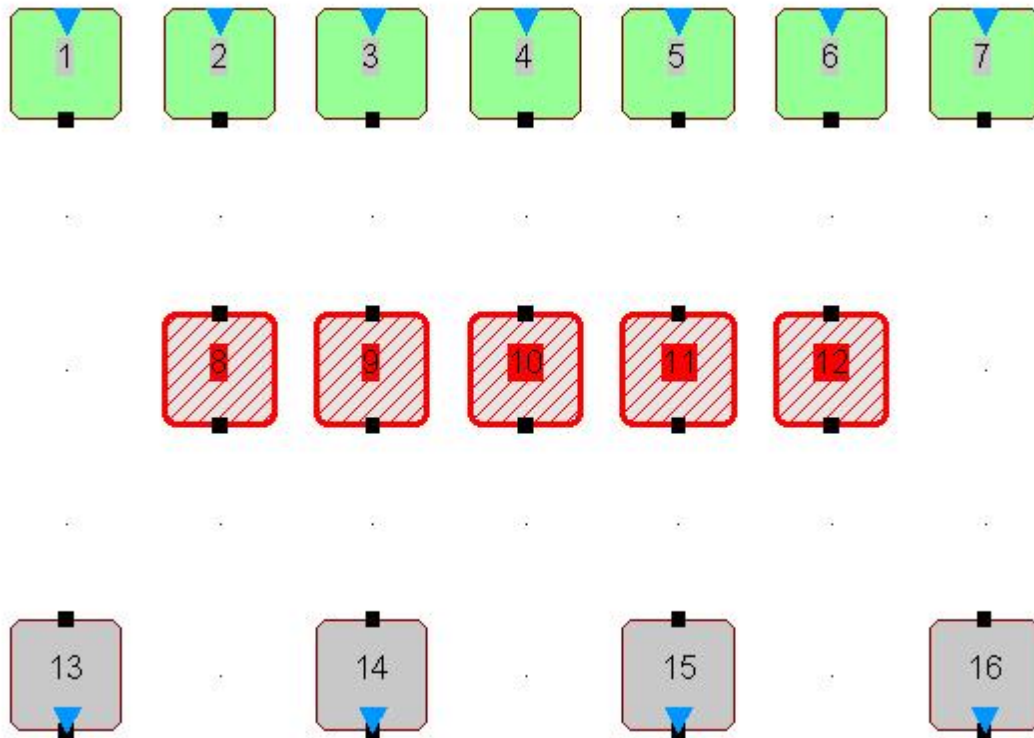
If you want to cancel the connection action just press <ESC> on the keyboard.

During connecting neurons you can also [navigate](#) to regions of the drawing currently not visible without having to cancel the current connect operation.

Connect FROM Extra Selection (FULL)

To connect two different groups of neurons you can use the Connect FROM Extra Selection feature. With this functionality it is very easy to add multiple connections between whole layers of a net:

- Select the source neurons FROM where the links shall go out
- Apply the [Extra Selection](#) state to them
- Select the target neurons to where the links shall go:

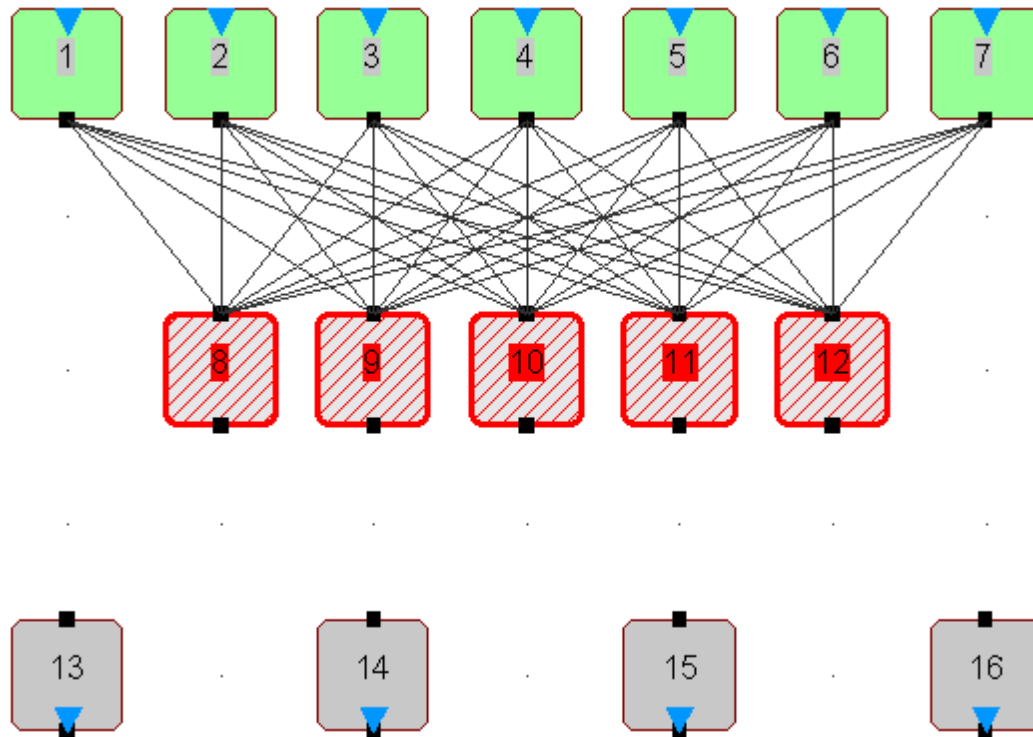


- Click on the tool bar symbol



(Or select <Edit><Connect FROM Extra Selection> from the main menu or from the context menu that will appear if you perform a right mouse click on one of the selected neurons).

The links are created:



Note: New links are always created with default properties. The default properties can be set by choosing <Edit><Default Properties...> from the main menu.

Note: You can even select neurons as target that are additionally Extra Selected (as source). This will cause round robin links to be created from the outputs of the Extra Selected neurons to their own inputs and to the inputs of all other selected neurons.

Tip: If you now immediately click on the tool bar symbol

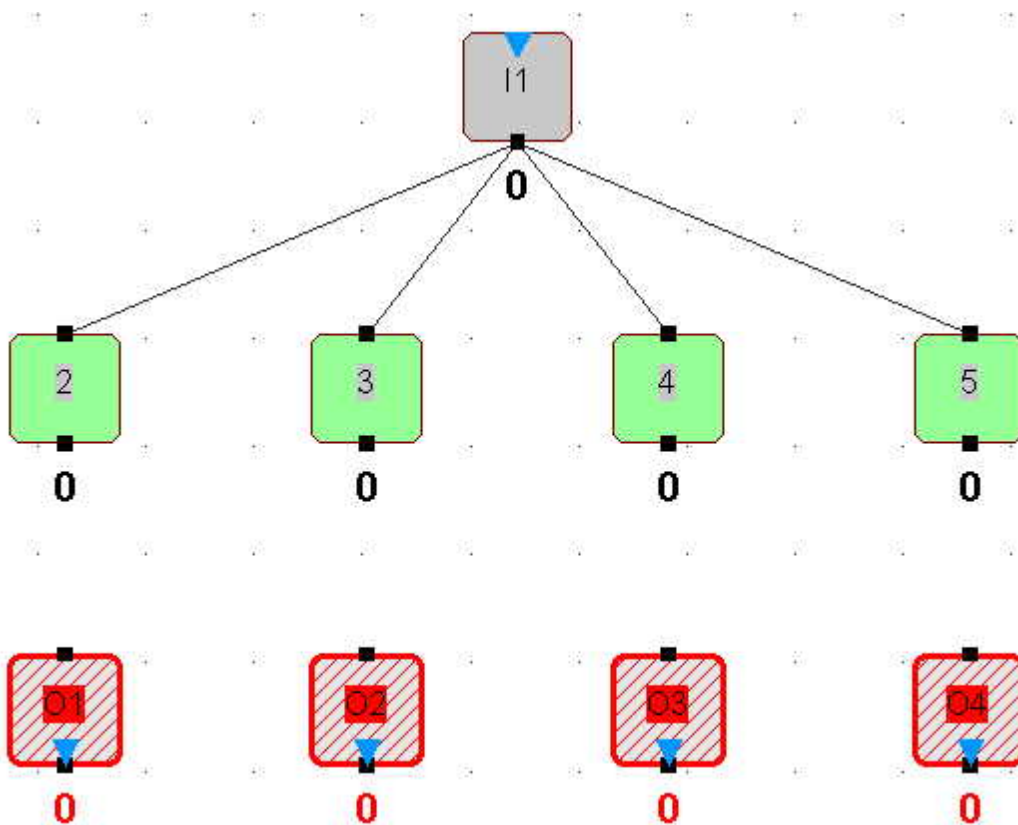


you automatically have the second layer of the net Extra Selected and thus can repeat the action to the next layer of the net very quickly.

Connect FROM Extra Selection (RANDOM)

To connect two different groups of neurons you can use the Random Connect FROM Extra Selection feature. With this functionality it is very easy to add multiple connections between whole layers of a net:

- Select the source neurons FROM where the links shall go out
- Apply the [Extra Selection](#) state to them
- Select the target neurons to where the links shall go:

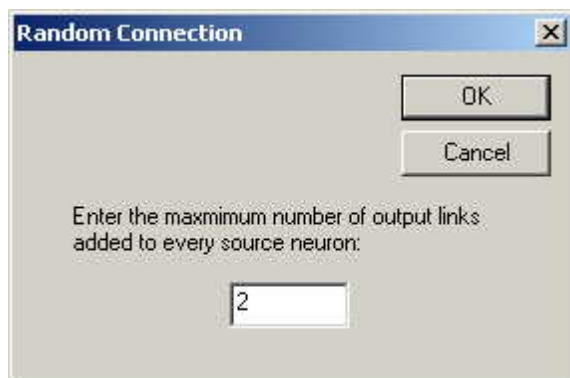


- Click on the tool bar symbol



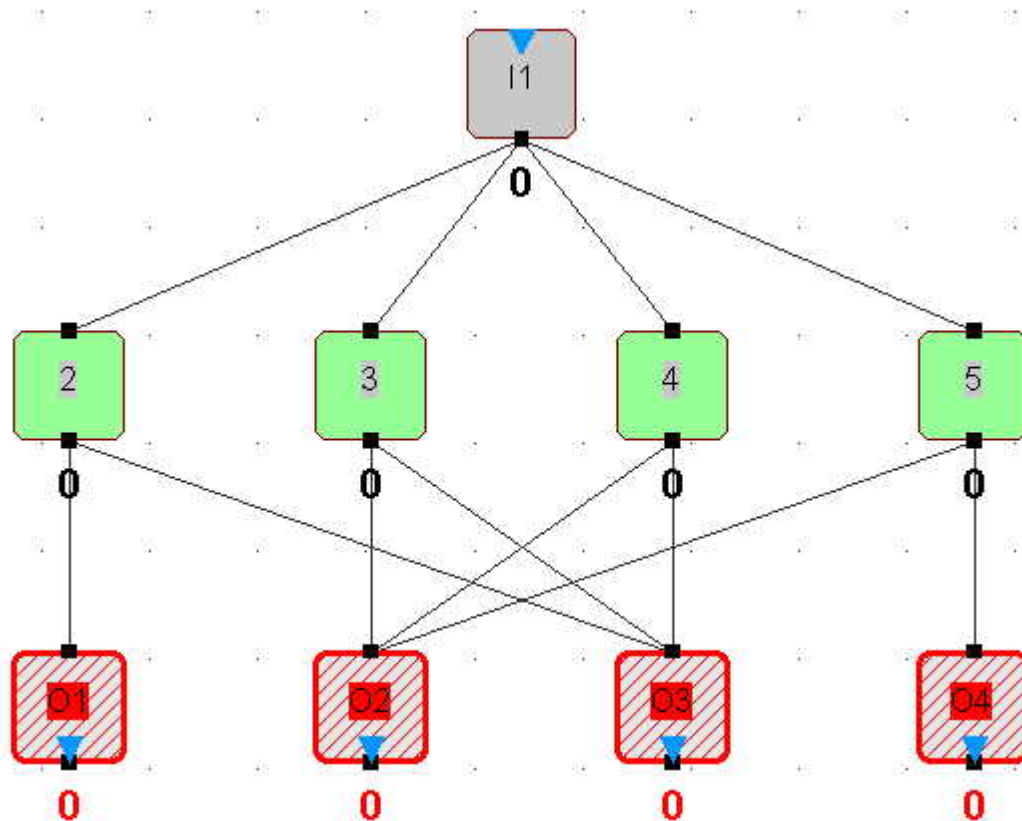
(Or select <Edit><Random Connect FROM Extra Selection> from the main menu or from the context menu that will appear if you perform a right mouse click on one of the selected neurons).

The following dialog will open.



You must now choose the maximum number of output links that shall be created for every source neuron. The algorithm will try to randomly connect the Extra Selected neurons to the selected neurons and not add more links than specified. It may happen that fewer links are added than the maximum number specified: If there is already a connection in place to a target neuron then no link is added. For every source neuron the specified maximum number of attempts are performed.

The links are created when clicking OK in the dialog:



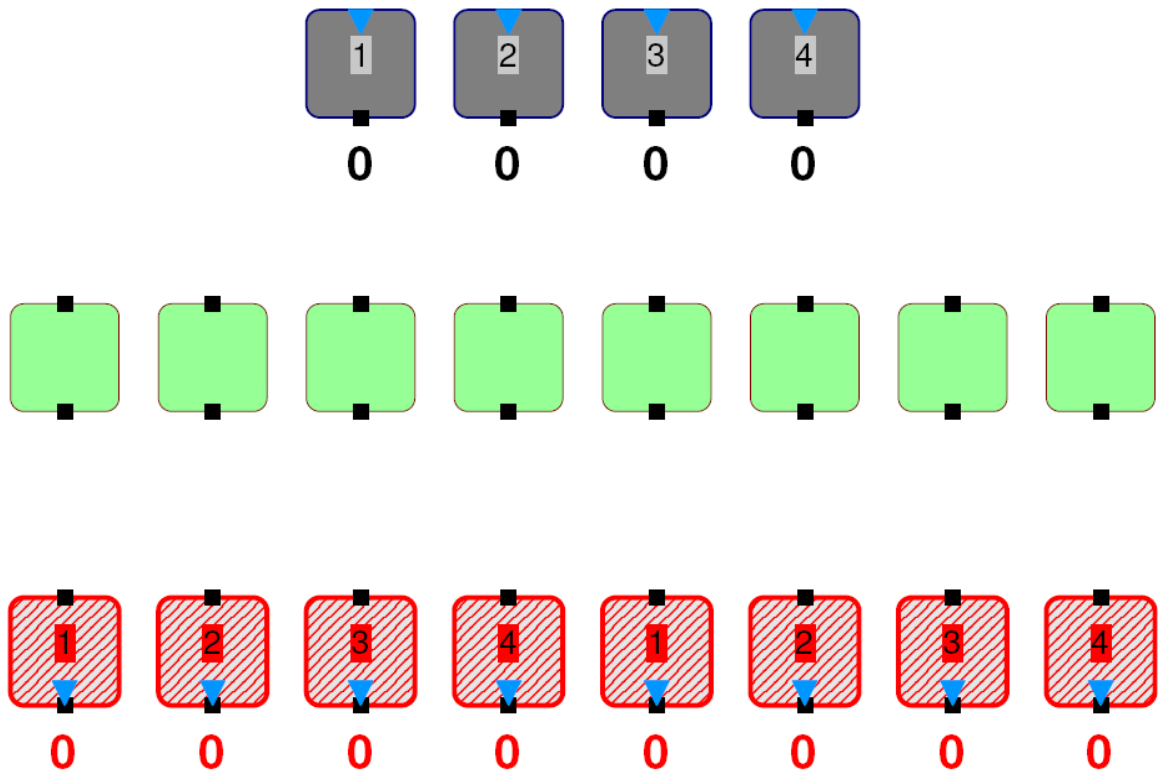
Note: New links are always created with default properties. The default properties can be set by choosing <Edit><Default Properties...> from the main menu.

Note: You can even select neurons as target that are additionally Extra Selected (as source). This will cause round robin links to be created from the outputs of the Extra Selected neurons to their own inputs and to the inputs of the other selected neurons.

Connect FROM Extra Selection (1:1)

To connect two different groups of neurons you can use the 1:1 Connect FROM Extra Selection feature. With this functionality it is very easy to add multiple connections between whole layers of a net:

- Select the source neurons FROM where the links shall go out
- Apply the [Extra Selection](#) state to them
- Select the target neurons to where the links shall go:

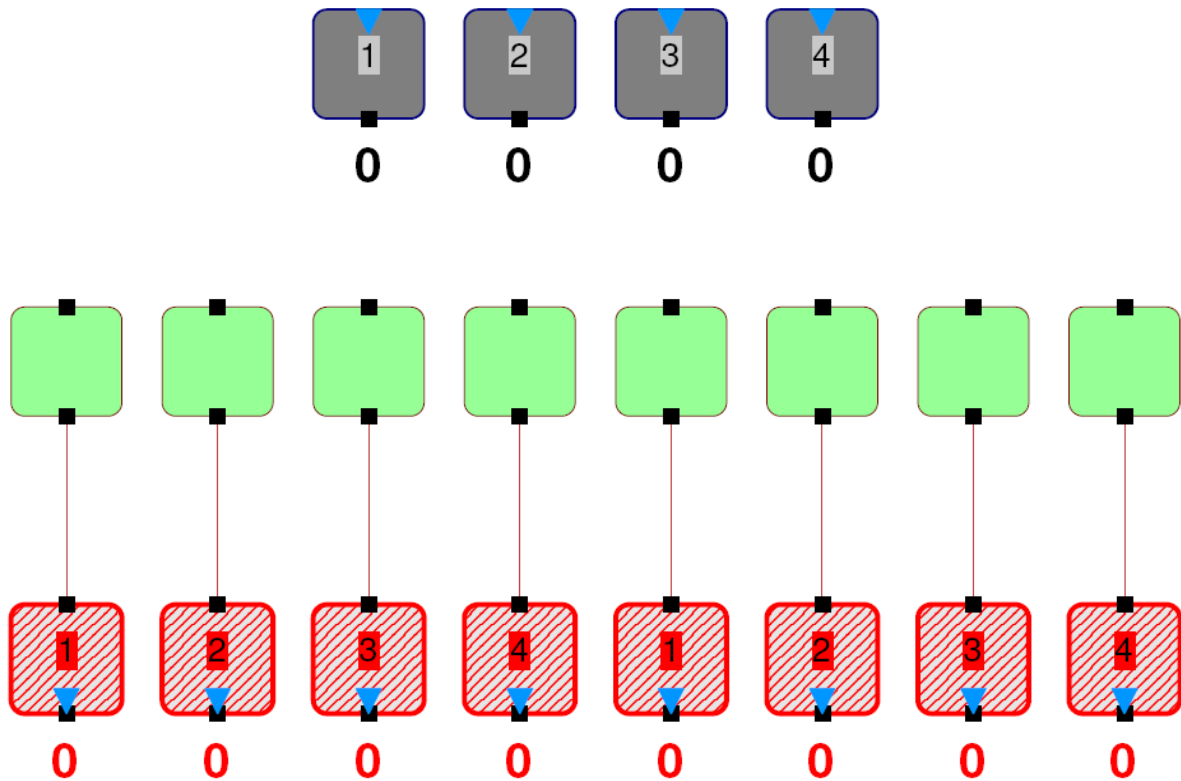


- Click on the tool bar symbol



(Or select <Edit><1:1 Connect FROM Extra Selection> from the main menu or from the context menu that will appear if you perform a right mouse click on one of the selected neurons).

The links are created:



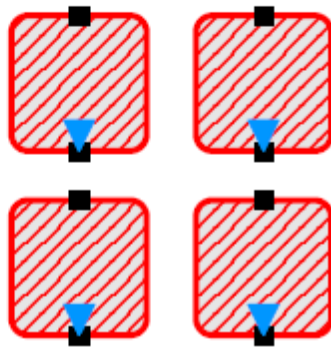
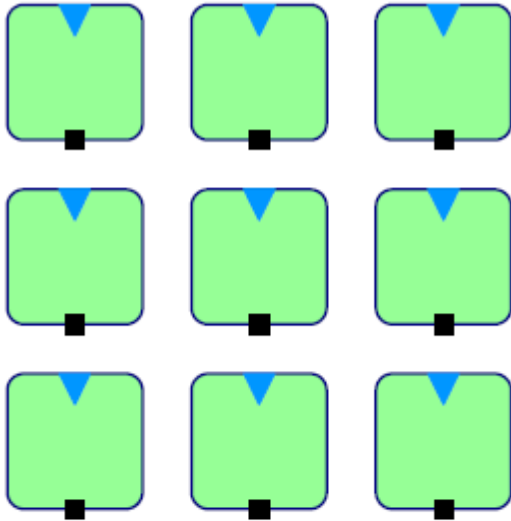
Note: New links are always created with default properties. The default properties can be set by choosing <Edit><Default Properties...> from the main menu.

Note: You can even select neurons as target that are additionally Extra Selected (as source). This will cause round robin links to be created from the outputs of the Extra Selected neurons to their own inputs.

Connect FROM Extra Selection (Matrix based)

To connect two different matrix (rectangular) shaped groups of neurons in a user defined way you can use the Matrix Connect FROM Extra Selection feature. With this functionality it is very easy to add multiple connections between whole neuron matrices of a net. The functionality provides user defined connection schemes by allowing the user to specify how sub groups within the selected neuron matrices are identified for interconnection. The sub groups within a matrix can also be specified to have an overlap. Thus, a neuron within a matrix can be part of more than one sub group for the connection scheme.

- Select the source neurons FROM where the links shall go out
- Apply the [Extra Selection](#) state to them
- Select the target neurons to where the links shall go:



- Click on the tool bar symbol



(Or select <Edit><Matrix Connect FROM Extra Selection> from the main menu or from the context menu that will appear if you perform a right mouse click on one of the selected neurons).

The following dialog will open.

Matrix Connection Specification

Specify below how the sub groups shall be chosen to interconnect the current Selection and the current Extra Selection.

The connections will be arranged using rectangular neuron groups with the specified dimensions in neurons. Additionally you can specify if the groups shall overlap each other and if so, then how big the overlap area shall be.

Selection Grouping - Matrix dimensions = 2 x 2 neurons

Group X Size: Group Y Size:

☒ No Group Overlap
☐ Overlap Groups By: Neurons

The current settings result in 4 connection groups for this matrix

Connect FROM:

Extra Selection Grouping - Matrix dimensions = 3 x 3 neurons

Group X Size: Group Y Size:

☐ No Group Overlap
☒ Overlap Groups By: Neurons

The current settings result in 4 connection groups for this matrix

The matrix connection is feasible.

You must now specify how the sub groups within the neuron matrices shall be chosen to make the connections. Every sub group within the Extra Selection matrix will be fully connected to the corresponding sub group in the selection matrix.

Note that the upper area of the dialog specifies the grouping within the selection matrix while the lower area specifies the grouping for the Extra Selection matrix.

For every matrix the dimensions are given in the grouping box headline. In the bottom of every grouping box the number of resulting neuron groups for the corresponding matrix is given. If grouping for a matrix is not possible with the currently adjusted settings then a corresponding error text is stated there instead. When the settings are changed in the dialog the result messages in the bottom of the grouping boxes update automatically on the fly. Thus, you can see the effects of your changes immediately.

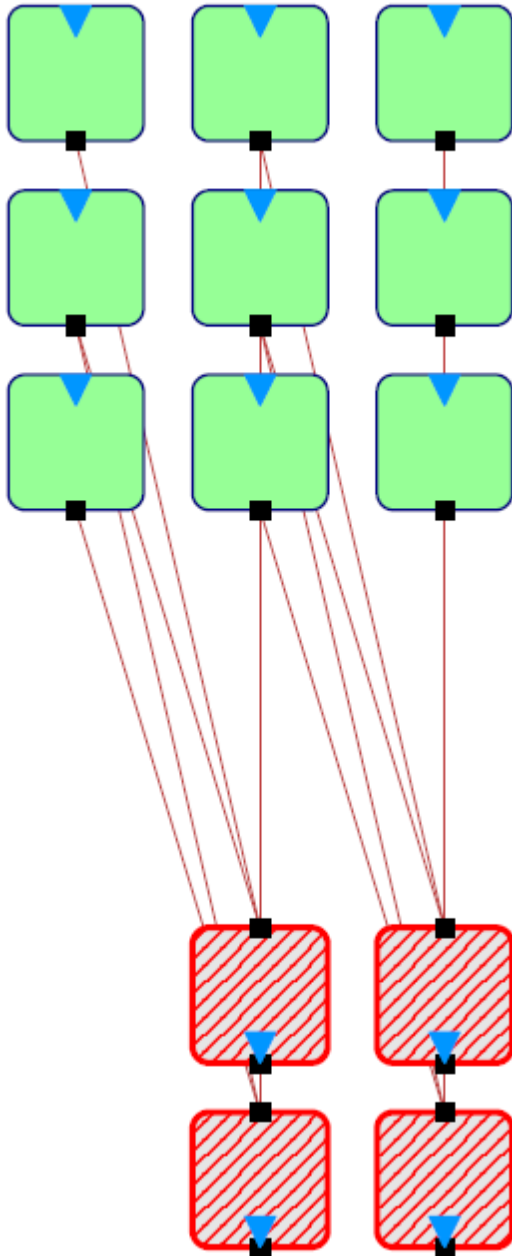
If both the grouping boxes show the same resulting number of identified groups then the matrix interconnection is feasible which is then stated in the bottom text line of the dialog as can be seen in the above picture.

In the given example the Selection neuron matrix is divided in sub groups with dimensions of 1 x 1 neurons which means that for the connection scheme the matrix is interpreted as consisting of single neurons or - to

put it the other way round - every neuron of this matrix represents a connection group.

For the Extra Selection the example chooses another grouping scheme: Every group shall consist of a square of 2 x 2 neurons and the groups shall overlap on their borders by 1 neuron. This results in four neuron groups for the Extra Selection matrix as identified in the bottom of the Extra Selection grouping box.

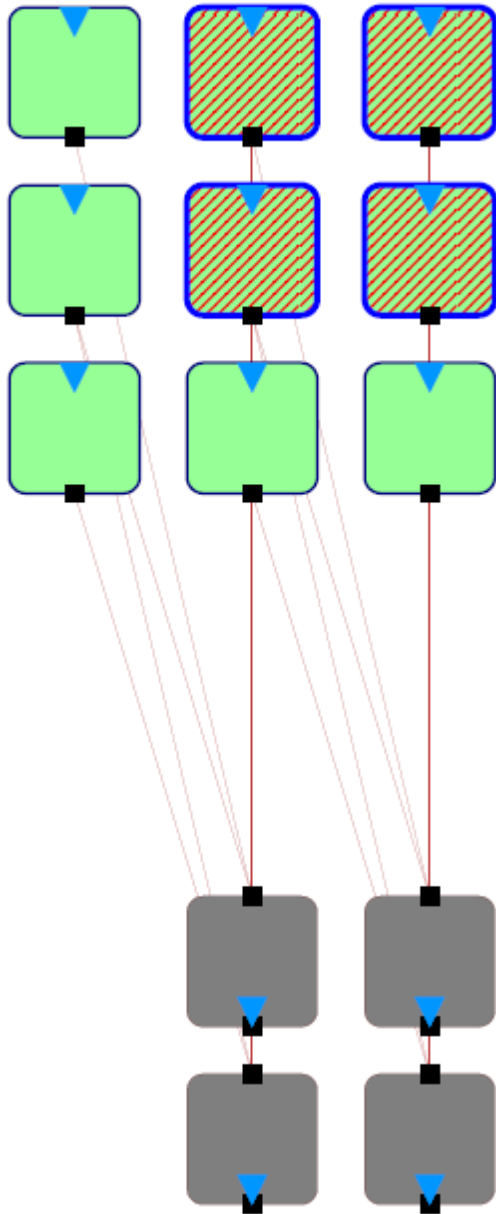
Since this is the same group count as identified for the Selection matrix the connection is feasible. A click on OK establishes the links between the neuron groups:



Note: New links are always created with default properties. The default properties can be set by choosing <Edit><Default Properties...> from the main menu.

Note: You can even select neurons as target that are additionally Extra Selected (as source). This will cause round robin links to be created from the outputs of the Extra Selected neurons to their own inputs and to the inputs of all other selected neurons according to the grouping specifications.

If you want to check how the connections have been set up you can use MemBrain's feature to select next or previous neurons in the net, respectively: Select one of the output neurons in the net and then choose <Edit><Select...><Select Previous Neurons> so select the neurons that are connected to the input of the currently selected neuron. In the example below the upper right output neuron had been selected before the <Edit><Select...><Select Previous Neurons> command was executed. According to the grouping specification during the matrix connection a group of 2 x 2 neurons is connected to the inputs of the output neuron.



Connect TO Extra Selection (FULL)

The feature Connect TO Extra Selection exactly works like Connect FROM Extra Selection, just the other way round: Source neurons are the selected neurons which are connected TO the Extra Selected neurons' inputs. Both methods can be used to reach the same result. It's just a question of what you prefer to work with.

The corresponding tool bar item is



So please refer to [Connect FROM Extra Selection \(FULL\)](#) for details on how the command works as it is almost the same.

Connect TO Extra Selection (RANDOM)

The feature Connect TO Extra Selection exactly works like Connect FROM Extra Selection, just the other way round: Source neurons are the selected neurons which are connected TO the Extra Selected neurons' inputs. Both methods can be used to reach the same result. It's just a question of what you prefer to work with.

The corresponding tool bar item is



So please refer to [Connect FROM Extra Selection \(RANDOM\)](#) for details on how the command works as it is almost the same.

Connect TO Extra Selection (1:1)

The feature Connect TO Extra Selection exactly works like Connect FROM Extra Selection, just the other way round: Source neurons are the selected neurons which are connected TO the Extra Selected neurons' inputs. Both methods can be used to reach the same result. It's just a question of what you prefer to work with.

The corresponding tool bar item is



So please refer to [Connect FROM Extra Selection \(1:1\)](#) for details on how the command works as it is almost the same.

Connect TO Extra Selection (Matrix based)

The feature Connect TO Extra Selection exactly works like Connect FROM Extra Selection, just the other way round: Source neurons are the selected neurons which are connected TO the Extra Selected neurons' inputs. Both methods can be used to reach the same result. It's just a question of what you prefer to work with.

The corresponding tool bar item is



So please refer to [Connect FROM Extra Selection \(Matrix based\)](#) for details on how the command works as it is almost the same.

Add Outputs to Selection

Note: For this functionality to be available the link display option of MemBrain must be enabled: Check that the option <View><Show Links> in MemBrain's main menu is enabled!

You can add an outgoing link to every selected neuron by clicking the tool bar button



or selecting the command <Insert><Output Links> from the main menu or the context menu that appears by a right click on selected neurons.

All open ends of the created links end at the mouse cursor and can be connected to the input of another neuron just the same way as with a [single link](#).

After connecting the links automatically new links will be created to connect to the next neuron. Press <ESC> on the keyboard to cancel connecting the latest created links.

During connecting neurons you can also [navigate](#) to regions of the drawing currently not visible without having to cancel the current connect operation.

Add Inputs to Selection

Note: For this functionality to be available the link display option of MemBrain must be enabled: Check that the option <View><Show Links> in MemBrain's main menu is enabled!

You can add an input link to every selected neuron by clicking the tool bar button



or selecting the command <Insert><Input Links> from the main menu or the context menu that appears by a right click on selected neurons.

All open ends of the created links end at the mouse cursor and can be connected to the output of another neuron just the same way as with a [single link](#).

After connecting the links automatically new links will be created to connect to the next neuron. Press <ESC> on the keyboard to cancel connecting the last created links.

During connecting neurons you can also [navigate](#) to regions of the drawing currently not visible without having to cancel the current connect operation.

Interconnect all neuron layers downwards

It is possible to interconnect all layers downwards in a neural net:

Select <Edit><Interconnect all layers downwards> or click the corresponding toolbar button:



Note1: This option is only enabled if your net contains at least one input and one output neuron.

Note2: If you have placed multiple unresolved neurons on the screen in several graphical layers then these layers will become interconnected in their graphical order on the screen from top to bottom. The first of these layers will get its inputs connected to either the input layer of the net (if no hidden layers already present) or to the outputs of the last hidden layer in the net.

Link Model And Operation

Links in MemBrain have two major properties that define their functionality:

- The Weight
- The (Logical) Length

The weight of a link defines how the signal magnitude is transported along the link:

The output signal of the neuron on the input side of the link is multiplied by the weight of the link. This is the value that will appear on the input of the neuron at the other end of the link. A link weight can even be negative and is not limited to a certain range. Thus a neuron cannot only activate but also inhibit another neuron depending on the weight of the link between the two neurons.

The Logical Length of a link defines how many calculation steps it takes to propagate a signal from the input end of the link to its output end. When MemBrain simulates a net then it repetitively performs one calculation step on every neuron which is called one "Think Step" on the net in MemBrain. When a neuron has calculated its output value (see [here](#) for details) it "pushes" this value into all of its output links causing all signals moving forward one step along the links. This means that a link with the (logical) length of 1 (minimum allowed value) propagates signals on its input side immediately to its output end so that no delay occurs.

Values bigger than 1 cause signals to be delayed along the links which brings time variant behaviour into your nets. You can also visualize this procedure by activating the the display of so called [Activation Spikes](#) on the links.

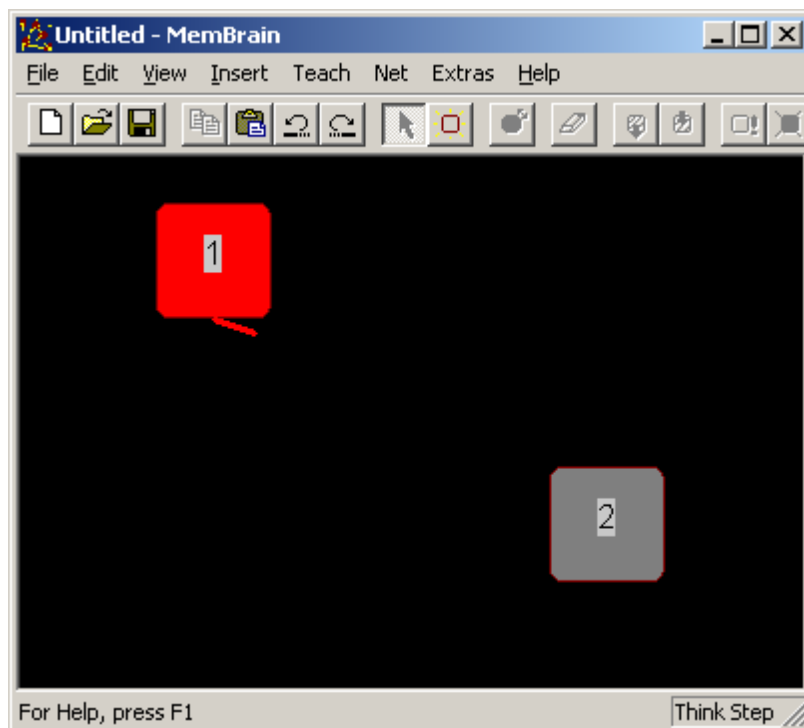
You can change the weight and the length of links together with their other [properties](#).

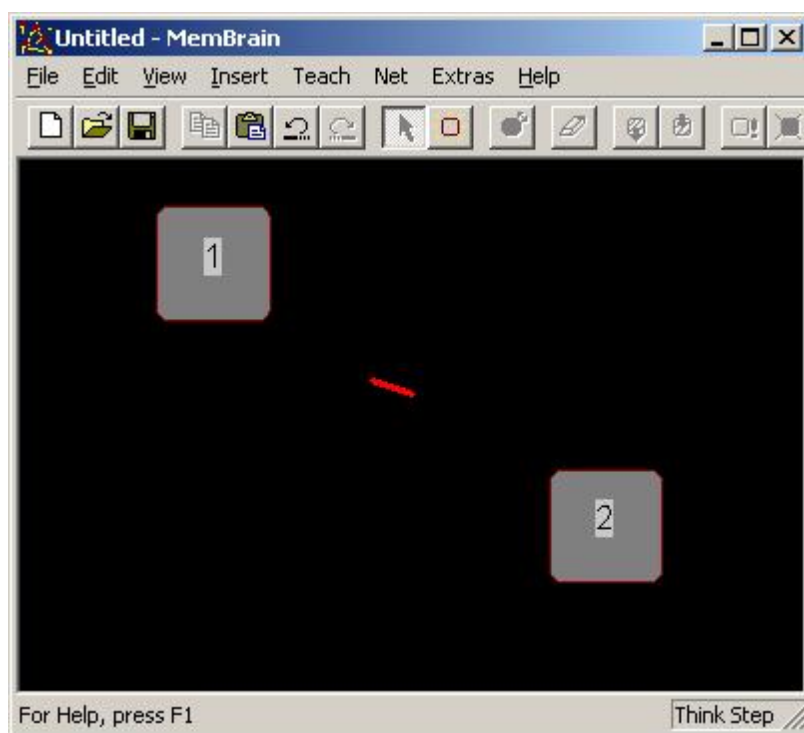
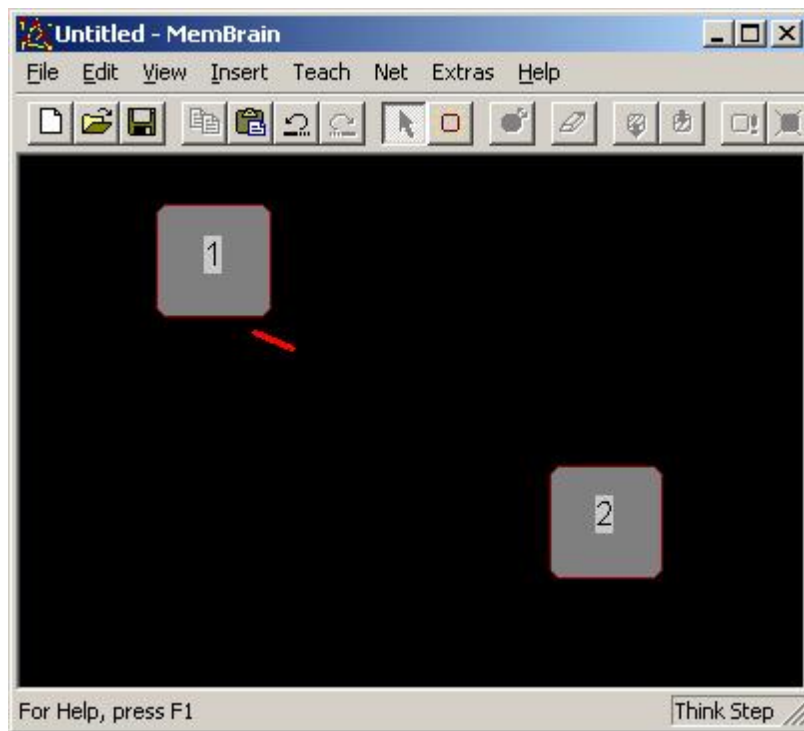
Additionally MemBrain provides a feature which allows you to set the logical length of some or all of the links according to their geometric length. Click [here](#) to learn more about that.

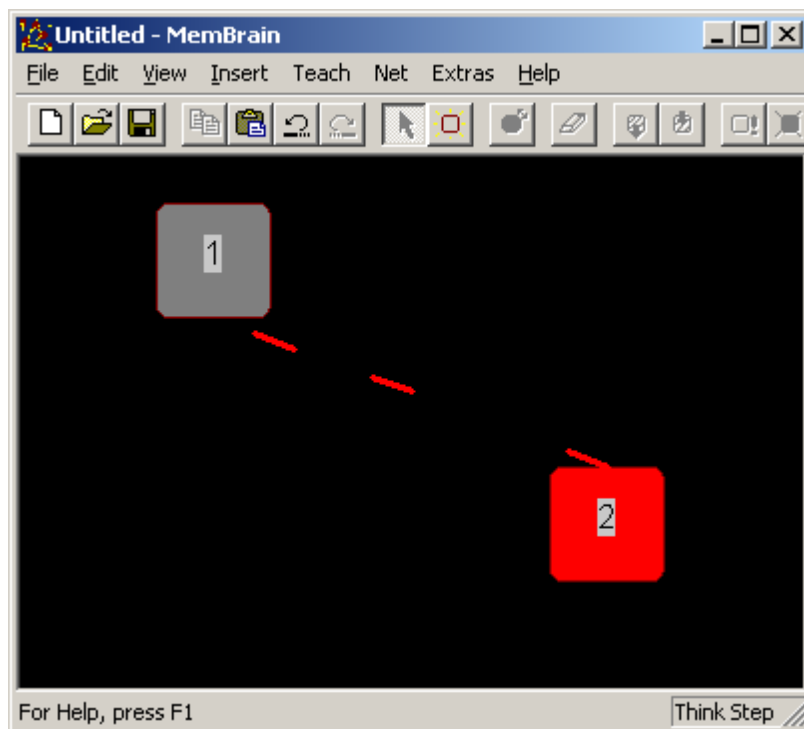
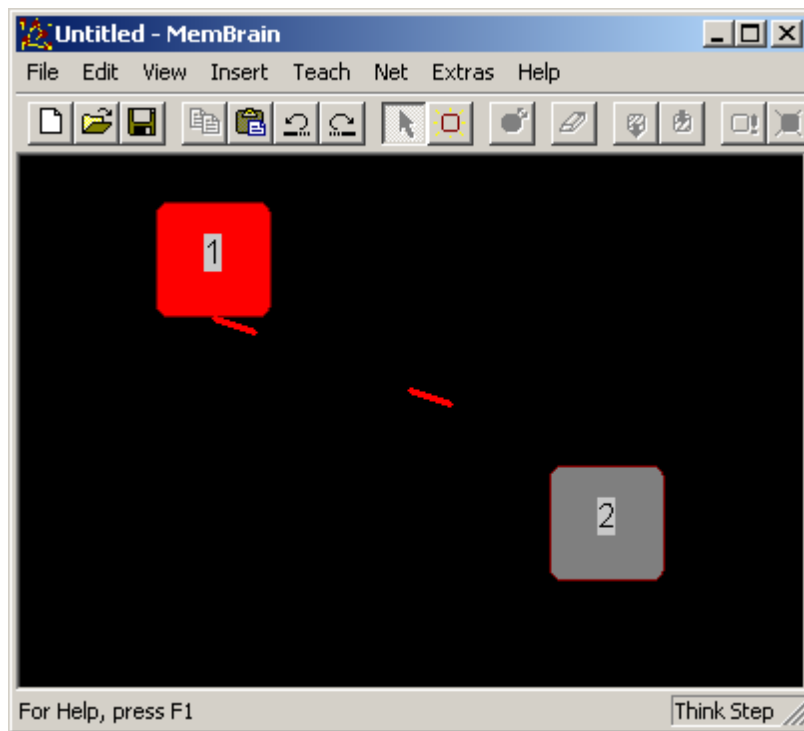
Activation Spikes Along Links

The activation spikes that flit along the links when MemBrain simulates the operation of a neural net can be visualized by activating the option <View><Show Activation Spikes on Links>.

Every time MemBrain performs one so called <Think Step> on the net the activation spikes will wander one link segment towards the input of the neuron connected to the output end of the link as the following simple example sequence illustrates.







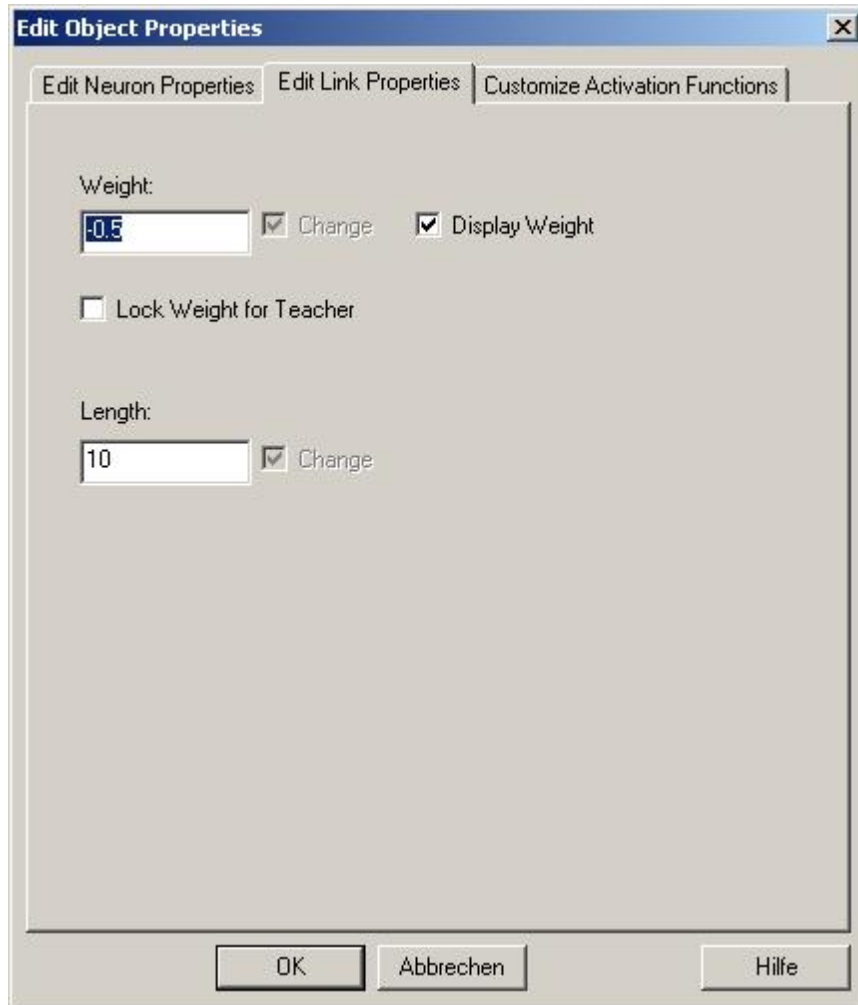
Changing Link Properties

To edit the properties of one or more links select them with the mouse and do one of the following.

- Right click on one of the selected links and choose <Properties> from the context menu
- Select <Edit><Properties> from the main menu
- Double click on one of the selected links (also works with single links that are not yet selected).
- Hit <ENTER> on the keyboard.

Note: For more information on how to select objects on the screen see [here](#).

The following dialog will open (change to the appropriate tab if necessary).



If you are not sure how the properties of a link influence its behaviour then please refer to the chapter about MemBrain's [Link Model](#).

The fields and their meanings:

- Weight: The current weight of the link.
- Display Weight: If checked then the current weight of the link is displayed on the screen together with the link.
- Lock Weight for Teacher: When this box is checked then the weight of the link will neither be changed during teaching the net nor by the Randomize Net command. For more information about teaching a net click [here](#).
- Length: The logical length of the link which specifies how many calculation steps it takes to propagate the signal from the link start point to its end point

The check box 'Change' is only functional if the properties of more than one link are edited. It is

- Grayed and checked if the corresponding setting is identical for all selected links (same as if only one

link is selected).

- White and unchecked if the corresponding setting is different for the selected links. In this case the setting itself is grayed. If the box is checked with the mouse the setting gets editable and the new value will be applied to all selected links when the dialog is closed using <OK>. The box can be unchecked again. Then the corresponding settings are not changed.

For details on the other tabs of the dialog refer to either [Changing Neuron Properties](#) or [Activation Functions](#).

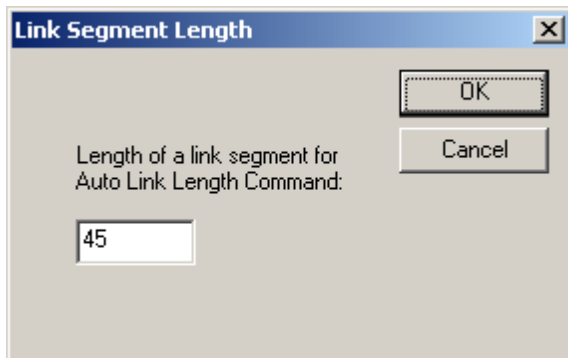
Click [here](#) for more information on how to add new links to the neural net.

Auto Link Length

MemBrain provides a feature to automatically set the logical length of links according to their geometric length.

To use this feature you first have to adjust the geometric length of a logical link segment by selecting <Extras><Link Segment Length...> from the main menu.

The following dialog will open:



The value you enter here is used to calculate the number of logical link segments a link consists of depending on its geometric length:

The geometric length of the link will be divided by the specified value and the resulting integer value will be the number of logical segments the link consists of (which is the logical length of the link).

As an orientation: The width of a neuron is 45 which is also MemBrain's default value for the link segment length.

Once you have adjusted the desired link segment length, select the links that shall be addressed by the Auto Link Length command and then choose the command <Extras><Auto Link Length> from MemBrain's main menu. All selected links now have their logical length adjusted in proportion to their geometric length on the screen.

Note: If you change the geometric length of a link on the screen by moving one of the neurons it is connected to, then this won't affect the logical length of the link. If you want the logical length of the link to be readjusted, too, then you have to select the link and again use the command <Extras><Auto Link Length>.

Reassign Links to Different Neurons

It is possible to re-assign the trained links of a group of neurons to another group of neurons.

This is useful if you want to [merge two already trained nets](#) for example: You then have to replace the input layer of one of the nets by the output layer of the other net without changing the weights of the links that come from the original input layer. I.e. you have to disconnect the links from the original input layer and connect them to the output neurons of the other net. This procedure is called *re-assigning links*.

See below how to:

[Re-assign Input links](#)

[Re-assign Output links](#)

Reassign Input Links

Here's the procedure how to re-assign the input links of a group of neurons to another group of neurons:

1. Apply [Extra Selection](#) to the group of neurons the input links shall be assigned to (in the following called *target neurons*)
2. Select the neurons from where the input links shall be disconnected (the *source neurons*). You will have to select the same number of neurons as for the Extra Selection i.e. every target neuron must have a corresponding source neuron.
3. Choose **<Extras><Re-assign Input Links FROM Selection TO Extra Selection>** from MemBrain's main menu.

All links that were formerly connected to the inputs of the source neurons are now connected to the inputs of the target neurons. All links have kept their properties including their current weight value.

When identifying matching target and source neurons MemBrain goes from left to right on the screen and line wise from top to bottom of the screen. I.e. the links of the top most and left most selected neuron will be re-assigned to the top most and left most extra selected neuron.

Reassign Output Links

Here's the procedure how to re-assign the output links of a group of neurons to another group of neurons:

1. Apply [Extra Selection](#) to the group of neurons the output links shall be assigned to (in the following called *target neurons*)
2. Select the neurons from where the output links shall be disconnected (the *source neurons*). You will have to select the same number of neurons as for the Extra Selection i.e. every target neuron must have a corresponding source neuron.
3. Choose **<Extras><Re-assign Output Links FROM Selection TO Extra Selection>** from MemBrain's main menu.

All links that were formerly connected to the Outputs of the source neurons are now connected to the outputs of the target neurons. All links have kept their properties including their current weight value.

When identifying matching target and source neurons MemBrain goes from left to right on the screen and line wise from top to bottom of the screen. I.e. the links of the top most and left most selected neuron will be re-assigned to the top most and left most extra selected neuron.

General Operations

This chapter contains general operation procedures used when editing neural networks with MemBrain.

You'll find the following sub chapters here.

- [Display Settings](#)
- [Dock and Show Toolbars](#)
- [Using the Grid](#)
- [Navigation in the Drawing Area](#)
- [Selecting Objects](#)
- [Deleting Objects](#)
- [Undo/Redo](#)
- [Copy/Paste](#)
- [Copy/Paste Neuron Activations](#)
- [Paste to Other Applications](#)
- [Start/Stop Simulation \(Auto Think\)](#)
- [Adjust MemBrain's Windows Process Priority](#)
- [Reset 'Don't Show Again' messages](#)

Display Settings

MemBrain offers several display options accessible via the <View> menu. A check mark beneath the options in the menu shows their current activation state.

- <View><Show Links>

If activated then links are displayed on the screen. Else the links and the connection ports of the neurons are hidden. You can only select links if they are displayed. Nevertheless you can interconnect neurons by all methods that do not require manual connection operations even if the links are currently not displayed. See [here](#) for details on how to interconnect neurons.

- <View><Show Activation Spikes on Links>

If activated then the activation spikes along the links are displayed. See [here](#) for details.

- <View><Show Fire Indicators>

If activated then neurons show a yellow dot on their output connector every time they fire (put out a signal $\neq 0$). The following pictures show a neuron that is currently firing. In the first picture the option <View><Show Links> is deactivated, so no output connector is shown. The second picture is the resulting display if the option is activated.



- <View><Show Winner Neuron>

If activated then the output neuron with the highest (normalized) activation is indicated after each Think Step as following:



This can be used to identify the winner neuron in a Self Organizing Map (SOM) for example.

- <View><Draw Links in Foreground>

If this option is set then the links are drawn in front of the neurons. Else they are displayed as to be behind the neurons. This setting also applies to the displayed activation spikes on the links.

- <View><Show Layer Info>

The Layer information for every neuron can be displayed and hidden by use of this menu item. See [here](#) for more information.

- <View><Use Display Cache>

When you activate this option then MemBrain prepares the display contents for the next screen update in an internal cache and blits them to the screen in one shot. This provides a flicker free display update but may be a little slower in some situations. Nevertheless sometimes it is better to not use the display cache because when working with large nets you might want to see the progress in display update in order to see what's going on.

- <View><Black Background (with Disp. Cache only)>

This option selects a black background for the display which can improve the visual appearance of some nets, especially when activation spikes are displayed and the net is simulated in [Auto Think](#) Mode. The option is only available when using the Display Cache.

- <View><Show Grid>

Shows/Hides the neuron placement grid. For more information on using the grid see [here](#).

- <View><Update View during Teach>

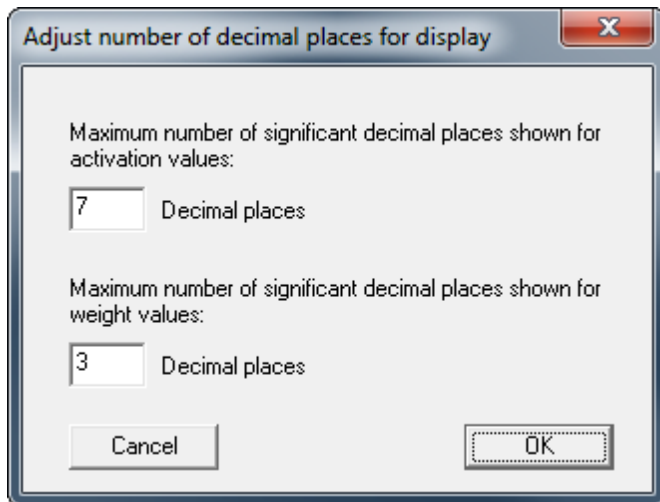
If checked then the view is updated during the teaching process after every lesson exercise of the net. For more details on how teaching is performed see [here](#).

- <View><Update View during Think>

If checked then the view is updated during the Auto Think process after every Think Step of the net. For more details on Auto Think see [here](#).

- <View><Adjust Precision for Display>

This option brings up the following dialog:

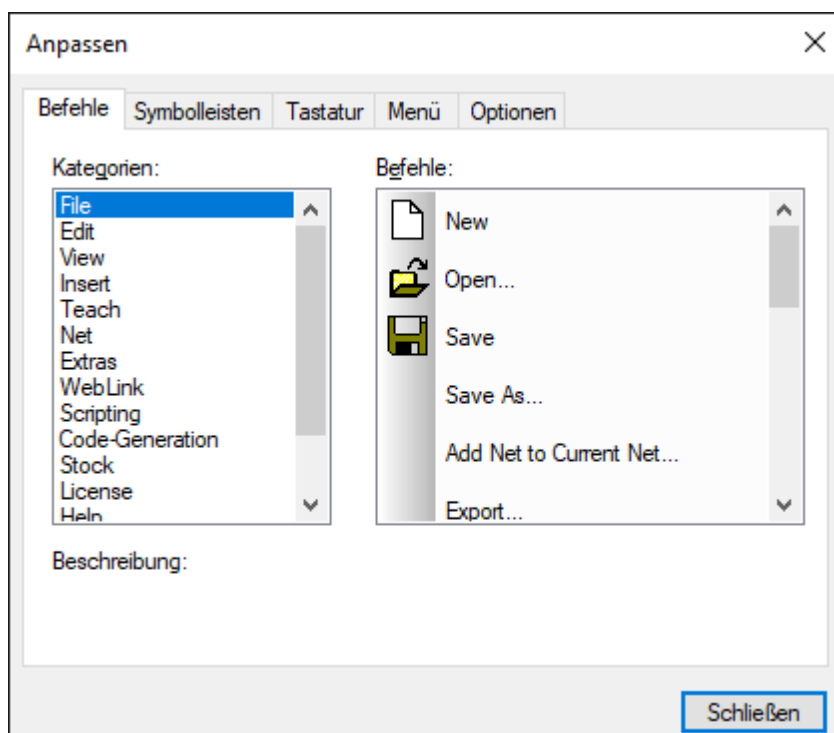


The number of decimal places displayed in MemBrain's main window for neuron activations and link weights can be adjusted here.

Customize menus, toolbars and app look

You can customize the basic application look via the menu item <View><Application Look>.

Customization of toolbars and menus is possible via <View><Toolbars and Docking Windows...><Customize...>:



If you want to show and re-arrange all toolbars automatically then select the menu command <View><Dock and Show all Toolbars> or press <Ctrl> + 'T' on the keyboard. This command will arrange all toolbars to be shown docked to the top border of MemBrain's main window.

Using the Grid

When using MemBrain you can choose to work on the basis of a placement grid for the neurons or to

operate with free neuron placement. To change the current setting use the command <View><Snap To Grid> from the main menu.

If the grid is displayed on the screen or not can be determined by a separate option: Use main menu's <View><Display Grid> command to display/hide the grid.

You can also set the width of the grid in a range from 5 to 500 units using <View><Set Grid Width...> As an orientation: The width of a neuron is 45.

There is also an option to realign all currently selected neurons to their nearest grid point: Use <Edit><Snap Selection to Grid> from the main menu.

Navigating In the Drawing Area

When holding down the SHIFT key on the keyboard the cursor changes to reflect a four fold arrow. While this arrow is displayed the visible rectangle can be moved across the drawing area: Additionally press the left mouse button, hold it down and move the mouse. This will feel like dragging the drawing area around with the mouse. Release the mouse button again and the drawing area will stop moving. You can repeat this action as long as you keep the SHIFT key pressed. Release the SHIFT key and the cursor will change to its previous shape and the last selected function will be active again. This functionality can be used at any time, no matter what other action is currently performed.

If you want to permanently activate this function then click the icon



on the tool bar. Click the icon again to return to the previously active function.

You can use the main menu function <View><Center> to get the center of the viewing area match the center of the drawing area.

Use the mouse wheel to scroll vertically in the drawing area at any time during operation.

It is also possible to zoom into the drawing area using either one of the following options:

- Right click and select <Zoom In> from the context menu. The view will be magnified and adjusted in a manner so that the point where the right click happened is located in the center of the new viewing rectangle
- Click on the tool bar symbol.



The center of the view will not change (Same functionality as <View><Zoom In> from the main menu. Or simply press <+> on the keyboard)

- Press and hold the <Ctrl> key and use the mouse wheel (push the wheel away from you)
- Click on the tool bar symbol



The cursor will change to reflect a cross hair. Select the desired viewing region by drawing a rectangle with the mouse, holding the left mouse button down. Release the mouse button and the view will be adjusted to the selected region. (Same functionality as <View><Zoom Rect> from the main menu)

- Click on the tool bar symbol



This will adjust the viewing area to fit the whole neural net. This might also lead to a zoom out action

depending on the currently viewable area. (Same functionality as <View><Zoom Fit> from the main menu)

Similarly it is possible to zoom out of the drawing area using the corresponding zoom out functions provided, e.g.



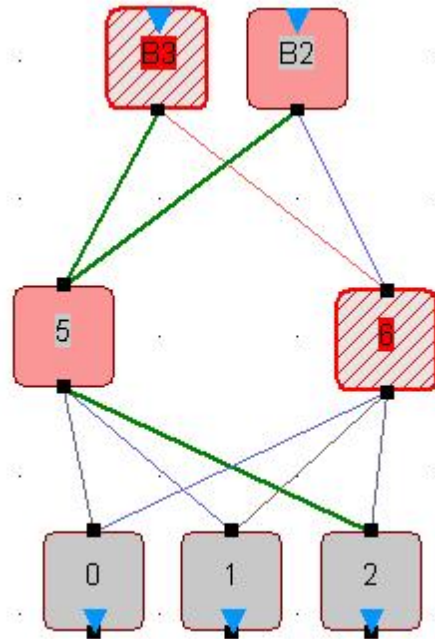
(Press and hold the <Ctrl> key and use the mouse wheel (pull the wheel towards you) Or simply press <-> on the keyboard)

Note that the current mouse cursor position is always displayed in the bottom of MemBrain's main window.

Selecting Objects

In order to perform operations with neurons and links in MemBrain it is often necessary to first select the target objects and then launch the operation.

Selected neurons are displayed with a red hatching, a selected link will appear more thick than normal and in green color:



The picture shows some selected neurons and links together with not selected ones.

Neurons and links in MemBrain can be selected by numerous methods:

- Left mouse click on an object:**
 A left mouse click on a neuron or link will select it and deselect all other objects. If you hold down the <CTRL> key on the keyboard while performing the action then the object's selection state is toggled without influencing the selection of the other objects. By this means single objects can be added or removed from the selection.
- Draw a selection rectangle:**
 Hold down the left mouse button and draw a selection rectangle over an area of interest. All objects that are completely included in the rectangle when releasing the mouse button will be selected. All others will be de-selected.
 If you hold down the <CTRL> key on the keyboard while performing the action then the selection states

of the enclosed objects is toggled without influencing the selection of the other objects. By this means multiple objects can be added or removed from the selection.

- Middle mouse click on an object:

If there are several objects that could potentially be hit by a mouse click you can use the middle mouse button to change the selection to the next alternative in reach of the mouse cursor. All other objects will be de-selected.

- "Paint Brush" selection:

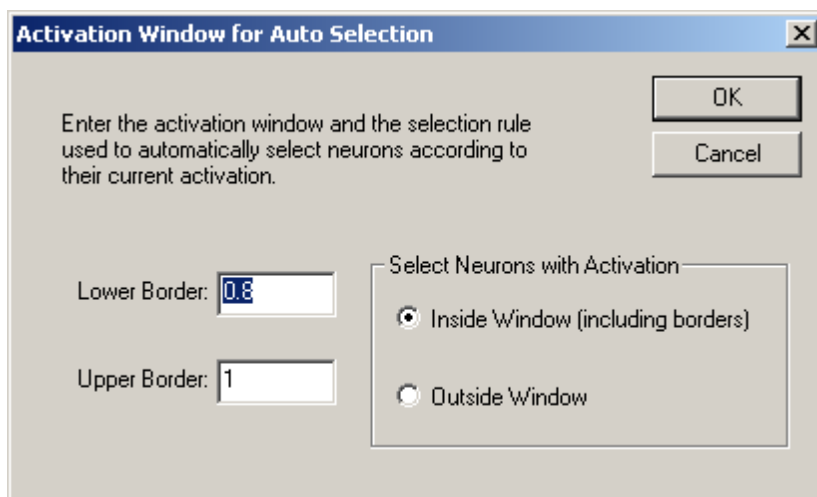
When holding down the <CTRL> key and the <SHIFT> key on the keyboard you can use the mouse like a paint brush to select objects: Hold down the left mouse button and move the mouse over the objects that shall be selected. All objects that are hit by the mouse on its way across the drawing area will get selected. Already selected objects will not be affected.

- Selecting the whole net:

If you choose <Edit><Select...><Select All> from the main menu or press <CTRL><A> on the keyboard then all objects in the drawing area get selected.

- Neuron Auto selection:

MemBrain also has possibilities to automatically select neurons by comparing their current activation to a certain window. Use <Extras><Neuron Auto Selection Threshold...> from the main menu to configure the feature. The following dialog will open:



You can enter a lower and an upper border for the activation window and select the selection method on the right hand side of the dialog. If you choose <Extras><Neuron Auto Selection (by Thresholds)> from the main menu then all neurons with an activation fitting the configured filter criteria are selected. All other neurons are de-selected.

Note: MemBrain uses the normalized (i.e. internally used) activation ranges here and NOT the user defined Normalization ranges.

- Selecting Round Robin Links only:

Sometimes it is useful to select all the Round Robin Links in a net. E.g. you might want to remove all Round Robin Links from a net but it might be difficult to get them all (and only them) selected. In this case choose <Edit><Select...><Select Round Robin Links> and all of them will automatically get selected. All other objects will become de-selected. A Round Robin Link is a link that connects the output and the input connector of one and the same neuron.

- Select Links between Selection and Extra Selection:

If you want to select all links that interconnect two groups of neurons in a certain logic direction then apply [Extra Selection](#) to one group and standard Selection to the other group. Then use the menu commands <Edit><Select...><Select Links FROM Extra Selection> or <Edit><Select...><Select Links

TO Extra Selection> depending on which direction of links you are interested in. All other objects will get de-selected.

- Selecting all output links of a group of neurons:

You can select all links connected to the outputs of a group of neurons by selecting the neurons and then execute <Edit><Select...><Select Output Links> or by pressing <Ctrl> + <Alt> + 'O' on the keyboard. There is also a toolbar symbol for this operation:



- Selecting all input links of a group of neurons:

You can select all links connected to the inputs of a group of neurons by selecting the neurons and then execute <Edit><Select...><Select Input Links> or by pressing <Ctrl> + <Alt> + 'I' on the keyboard. There is also a toolbar symbol for this operation:



- Selecting all destination neurons of a group of links:

It is possible to select all neurons that are destinations of a group of selected links. To do so select one or more links and then execute <Edit><Select...><Select Destination Neurons> or press <Ctrl> <Alt> + 'D' on the keyboard. There is also a toolbar symbol for this operation:



- Selecting all source neurons of a group of links:

It is possible to select all neurons that are sources of a group of selected links. To do so select one or more links and then execute <Edit><Select...><Select Source Neurons> or press <Ctrl> <Alt> + 'S' on the keyboard. There is also a toolbar symbol for this operation:

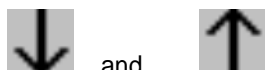


- Trace selection using the keypad:

You can follow the connection traces in your net using the <UP> and <DOWN> arrows on your keypad: Select a group of neurons or links, then press the DOWN arrow on your keypad. All connected links or neurons, respectively will get selected while the original selection is reset. The DOWN arrow traces through the net in a 'downwards' logic i.e. following the links from neuron outputs to inputs. Consequently, the UP arrow follows the links 'upwards' i.e. from neuron inputs to outputs.

Note: If you press and hold the <Ctrl> key on the keypad during the arrow operations then the trace will be added to the current selection instead of replacing it. Thus, using the <Ctrl> and the arrow keys you can select whole traces within a net.

Note: The menu commands <Extras><Trace Selection Downwards> and <Extras><Trace Selection Upwards> correspond to these commands as well as the toolbar buttons



Holding down the <Ctrl> key will only work in combination with the keypad commands, however.

- Browsing neuron-wise through a group of selected neurons:

If at any time a group of neurons is selected then pressing the LEFT and the RIGHT arrow on the keypad will select the upper left or the lower right neuron within the group, respectively. Then, pressing

the RIGHT or the LEFT arrow key will select the next or the previous neuron within the formerly selected group following a line-wise order from top to bottom within the group. This feature can be useful in combination with the Trace selection described above in order to follow a single connection trace through the net.

- Select the current Extra Selection

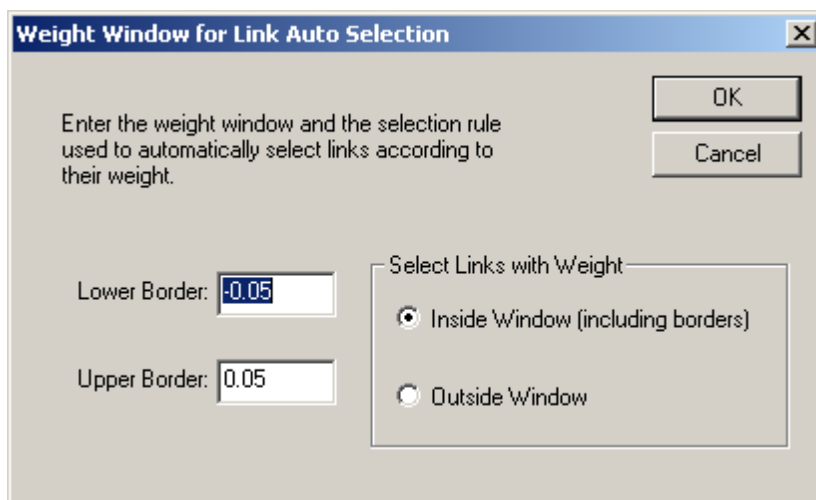
The current Extra Selected neurons can be selected by the command <Edit><Select Extra Selection> or by clicking on the corresponding tool bar button



All other objects currently selected will get de-selected.

- Link Auto Selection

MemBrain has possibilities to automatically select links by comparing their current weight to a certain window. Use <Extras><Link Auto Selection Threshold...> from the main menu to configure the feature. The following dialog will open:



You can enter a lower and an upper border for the weight window and choose the selection method on the right hand side of the dialog. If you choose <Extras><Link Auto Selection FROM Extra Selection (by Thresholds)> or <Extras><Link Auto Selection TO Extra Selection (by Thresholds)>, respectively, from the main menu then all links are selected whose weight fits the configured criteria and which route FROM the Extra Selection to the current selection or from the current selection TO the Extra Selection, respectively. All other links are de-selected.


- Select all members of a group:

Right click on a neuron or group which is member of a group. Then select <Select Group Members>. All members of the group are selected.

Note that this also works with members of multiple different groups at the same time.

- Select Convolutions

When one or more neurons or links are selected then MemBrain can automatically select all objects that share weight or threshold values with the selection.

Choose <Edit><Select...><Select Convolutions> or press the tool bar icon . All convolutional objects related to the current selection will be selected. All other objects will become de-selected.

Note that in MemBrain's status bar the number of objects of different types in the net as well as the number of selected objects of each type is always displayed.

Extra Selection

Although MemBrain has been designed to be as intuitive in handling as possible and thus makes use of many standard windows operation procedures when handling graphic objects like neurons and links, there is one feature that is very special to MemBrain:

It's the so called "Extra Selection" of neurons, [groups](#) or [proxy ports](#).

MemBrain's Extra Selection is a powerful feature that is mainly used when connecting different groups of neurons but there are also some other activities in MemBrain that make use of it.

The Extra Selection is a certain state neurons can be assigned in order to distinguish one selected group of neurons from another. For example if you want to connect all neurons of a certain group to all of another group then apply the Extra Selection to one group and the standard selection to the other group. Then click



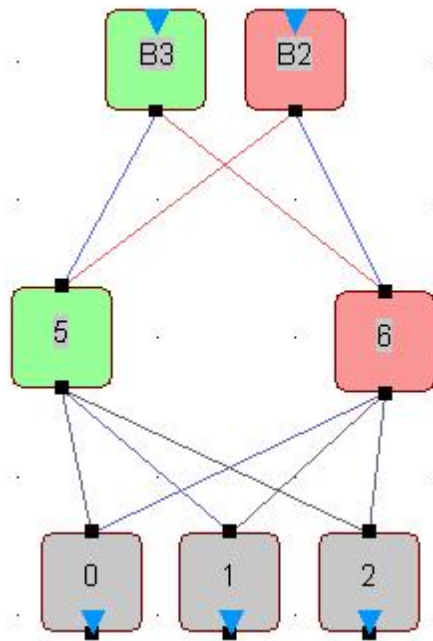
on the tool bar. All outputs of the Extra Selected group will be connected to all inputs of the neurons currently selected. This is just one method for connecting neurons. See [here](#) for more information on all of the possibilities to interconnect neurons in MemBrain.

To set neurons to the Extra Selected state first select them as usual (see [here](#) for details on selecting objects in MemBrain) and then do one of the following:

- Right click on one of the selected neurons and choose <Extra Selection> from the context menu.
- Choose <Edit><Extra Selection> from the main menu.
- Click on the tool bar symbol



The neurons will change their display style to reflect their Extra Selected state:



Extra Selection example: Neurons B3 and 5 are "Extra Selected".

Note: All neurons that are not currently selected when performing the "Extra Selection" command will be reset to normal state (no Extra Selection).

To reset the Extra Selection state of all neurons do one of the following:

- Right click somewhere in the drawing area and choose <Clear Extra Selection> from the context menu.
- Choose <Edit><Clear Extra Selection> from the main menu.
- Click on the tool bar symbol



You can also add neurons to the Extra Selection or remove them from the Extra Selection using the tool bar symbols



and



Note 1: Extra selection in MemBrain is always indicated by **green** color. This also applies to the tool bar buttons that perform commands in conjunction with the Extra Selection.

Note 2: A neuron can be both selected and Extra Selected at the same time!

Deleting Objects

In order to permanently delete objects like neurons and links just select them and then do one of the following.

- Press on the keyboard
- Select



from the toolbar

- Select <Edit><Delete> from the main menu

Note: When deleting neurons then all links connected to the deleted neurons are also deleted.

Undo/Redo

Generally, in MemBrain all actions that modify the network can be undone/redone using the Undo/Redo functionality:

- Select



respectively



from the tool bar

- Choose <Edit><Undo> or <Edit><Redo> from the main menu, respectively. Notice that in this case the type of the action that will be undone/redone is also listed in the menu.
- Press <Ctrl> + 'Z' or <Ctrl> + 'Y' on the keyboard

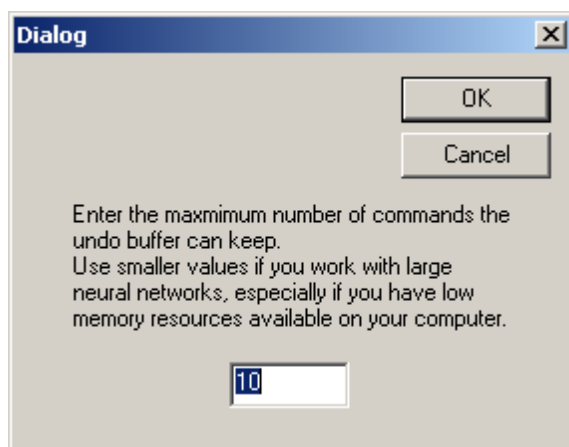
Note: Changes that are performed while teaching a net or while the net is in [Auto Think](#) mode are summarized under only one Undo/Redo action. So these changes can only be undone/redone as a whole. For more information on teaching neural nets in MemBrain click [here](#).

There currently is **no** Undo/Redo functionality implemented for the [Lesson Editor](#)!

The number of operations the undo/redo buffer keeps trace of can be adjusted between 0 and 100. The default value is 30 which is a good compromise between memory usage and flexibility when editing.

To adjust the size of the Undo/Redo buffer choose <Edit><Set Undo Buffer Size...>

The following dialog appears.



If you work with large neural networks and especially if you have low memory resources available on your machine it may make sense to reduce the size of the Undo/Redo buffer.

Copy/Paste

In MemBrain you can also use the copy/paste functionality to duplicate parts of a net:

- [Select](#) the parts of the net that shall be copied.
- Copy the selected objects to the clipboard by doing one of the following.

- Select



from the tool bar.

- Select <Edit><Copy> from the main menu
 - Press <CTRL><C> on the keyboard

- Paste the copied items from the clipboard by doing one of the following.

- Select



from the tool bar.

- Select <Edit><Paste> from the main menu
 - Press <CTRL><V> on the keyboard

- The pasted contents will be attached to the mouse cursor. Move the items to their target location in the drawing and left click on the mouse to place them.

Note 1: The pasted neurons will have the same names as the copied ones. Note that you can auto-generate names for all or a subset of the neurons in your net later on using the [Neuron Auto Naming](#) feature.

For more information about neurons in MemBrain click [here](#).

Note 1: When copying items to the clipboard, notice that **links between neurons are only copied if they:**

- **Are connected to SELECTED neurons on BOTH sides**

AND

- **Are SELECTED themselves**

Consequently, the option <View><Show Links> must be active in order to copy links

Copy/Paste Activations

If you want to copy and paste the current activations of a group of neurons to another group then just copy the source neurons to the clipboard as described [here](#) and then select

<Edit><Paste Neuron Activations> from the main menu or simply press <ALT><V> on the keyboard.

A set of neurons without names will be attached to the mouse symbolizing the pasted activations. Place the group of activations over a corresponding group of target neurons (groups must be identical in shape) and click with the left mouse button. The target neurons will adopt the pasted activations.

Note: All activations that are not exactly positioned over a target neuron when pressing the left mouse button are ignored.

Paste to Other Applications

Parts of a net can be pasted as bitmaps to other applications like your favorite word processor by simply copying the elements to the clipboard (see [here](#)) and then pasting them from the clipboard into the target document of another application (usually by pressing <Ctrl> + 'V' on the keyboard within the target application's window).

Start/Stop Simulation (Auto Think)

MemBrain is capable of performing a continuous simulation of a neural network that is, it repetitively performs Think Steps on the net and refreshes the display after every step.

This procedure is called <Auto Think> in MemBrain

Note: The display refresh during Auto Think can be suppressed via the [view menu](#). Thus, if your display does not refresh, please check the option <View><Update View during Think>.

To start or stop the Auto Think mode use the tool bar buttons



and

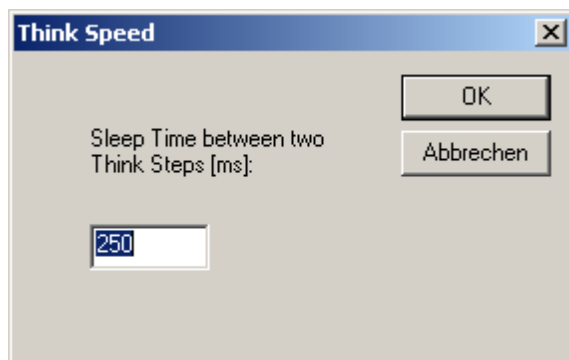


or use the corresponding menu commands:

<Net><Start Thinking (Auto)> and <Net><Stop Thinking (Auto)>

To configure the speed of the simulation you can choose

<Net><Set Simulation Speed...> which will open the following dialog.



The time value you can adjust here determines for how long MemBrain pauses the simulation after the display has been refreshed, i.e. between every two Think Steps. You can also change that value when the Auto Think mode is already running.

Alternatively, you can press the key <i> respectively <r> on the keyboard to increase or reduce the simulation speed in steps of ms. The current simulation sleep time between each two think steps is always displayed in MemBrain's status bar.

Note that during Auto Think you can still perform actions like moving or editing neurons or applying activations using the [Quick Activation](#) feature. Other features like deleting neurons are locked during Auto Think.

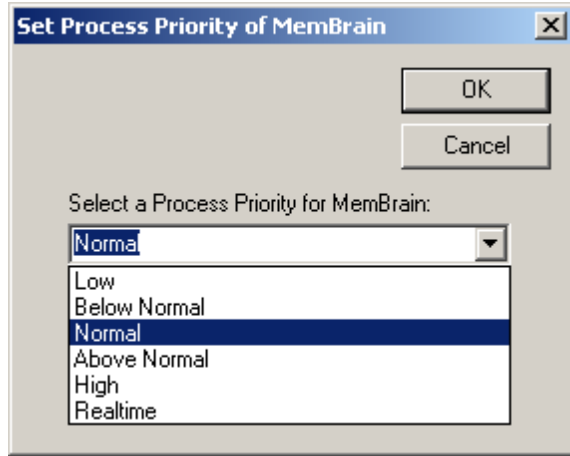
In the status bar MemBrain always displays the current number of Think Steps performed. You can reset that counter manually by selecting <Net><Reset Think Step Counter> from the main menu. In the beginning of every Auto Think or Auto Teach procedure this counter is reset automatically.

MemBrain Process Priority

In order to improve the resource management on your computer you can adjust MemBrain's process priority according to your needs. If you do lengthy operations in MemBrain (e.g. teaching) then you can reduce

MemBrain's process priority for example to gain free calculation time on your computer for other applications.

Using the same settings as in the Windows Task Manager you can choose among the following values for MemBrain's process priority.



This dialog opens if you choose <Extras><Set Process Priority...> from MemBrain's main menu.

You can change the process priority of MemBrain at any time during operation. The priority also is stored to file so you don't have to set it every time you start the application.

Note: MemBrain provides all process priorities available within Windows. However, the only two recommended ones are 'Normal' and 'Below Normal': If you feel that your other applications are slowed down or show delayed reaction on user input while MemBrain's performs lengthy operations then you should select the setting 'Below Normal' (Default). Else choose 'Normal'.

Reset 'Don't Show Again' messages

MemBrain shows some important hints during operation that are mostly interesting for beginners but which can get a bit annoying to more experienced users.

These messages all have a checkbox 'Don't show this dialog again' which - ones checked - suppresses the corresponding message in the future. However, it may be desired for one or the other reason to have the messages come up again. Because of this MemBrain supports the possibility to reactivate all of the messages again.

This is performed by the menu command <Extras><Reactivate All Don't Show Again Messages>.

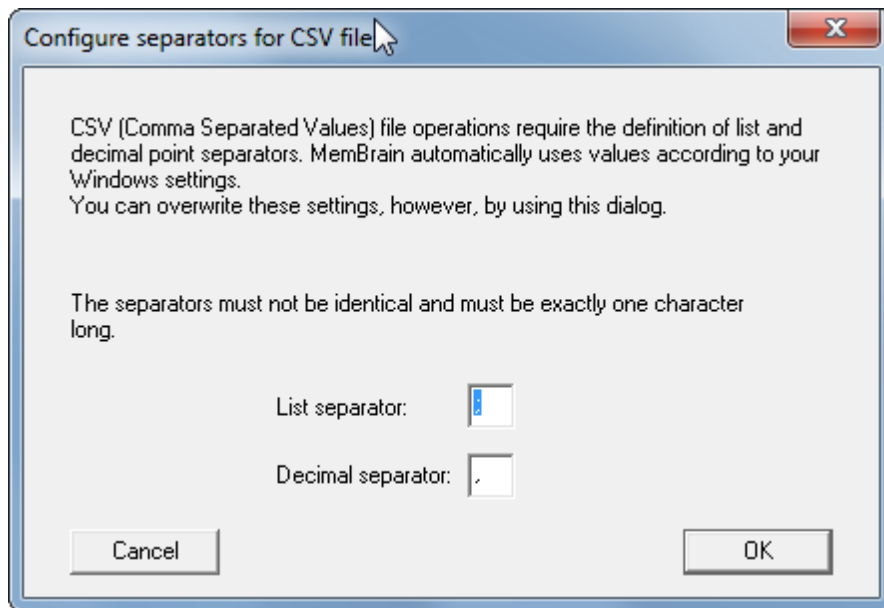
CSV file separator settings

For all operations which use CSV (Comma Separated Values) files, MemBrain needs to know which separator characters to use for parsing (during CSV import) and generation (during export) of CSV files. Normally, these values are taken from the Computer's so called 'locale' settings which means that the settings are used according to the operating system's installation language.

For German operating system language this normally means that the semicolon ';' is used for separating data columns ('list separator') and the comma ',' is used as decimal separator (i.e. to separate decimal places within number strings). For English or USA language settings the list separator normally is the comma ',' while the decimal separator is the point '.'.

If you need to change the separator settings of MemBrain for some reason then this can be performed via the main menu command <File><CSV file separator settings...>.

The following dialog will appear:



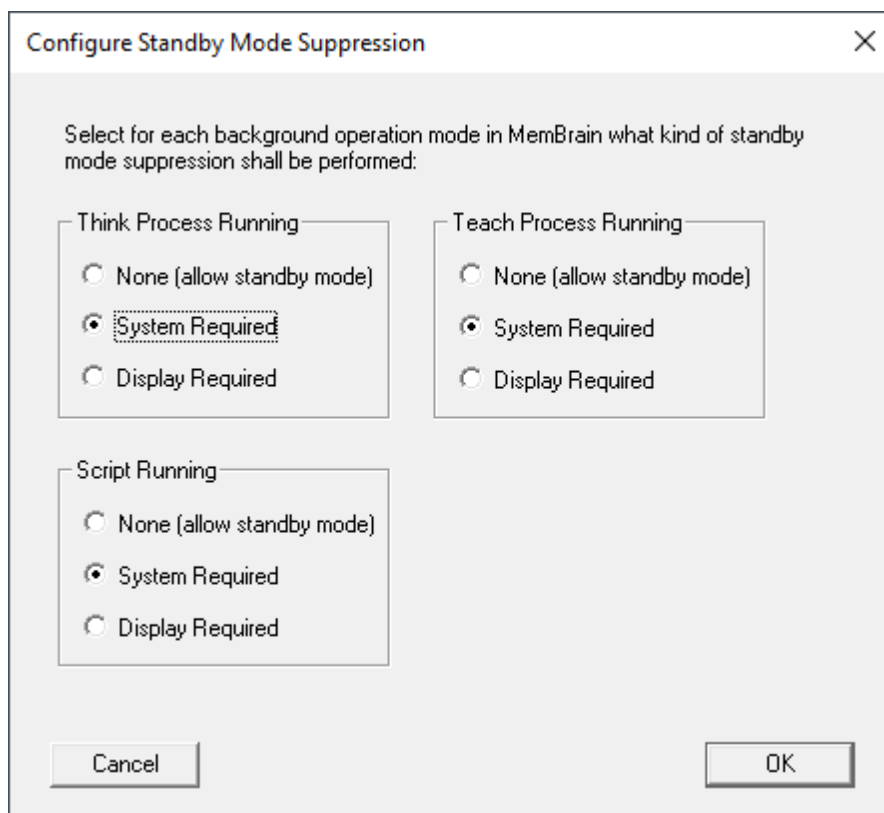
Simply adjust the separator characters as required and click <OK>.

MemBrain will remember these settings and re-load them automatically when started the next time.

Standby Mode Settings

You can adjust how MemBrain behaves when your computer attempts to enter standby mode while MemBrain is busy with background operations:

Select <Extras><Standby Mode Suppression...>. The following dialog will come up:



For the three categories:

- Think Process Running
- Teach Process Running

- Sript Running

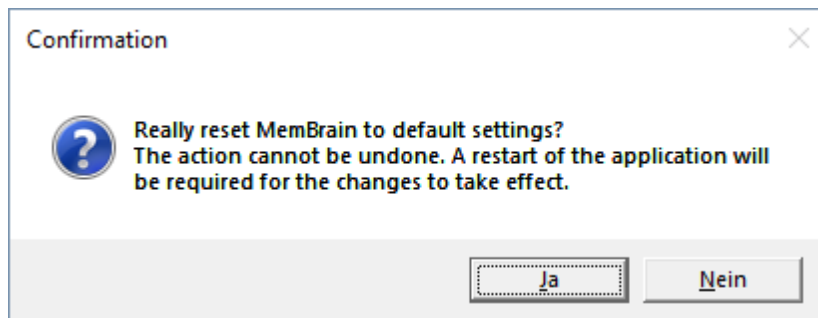
you can adjust whether MemBrain will suppress the standby mode of your computer and how:

- None --> Normal standby mode operation as adjusted in your WIndows settings
- System Required --> Will prevent the computer from entering standby mode. I.e. MemBrain will still continue with its background operations
- Display Required --> Like <System Required> but additionally, MemBrain will prevent that your display device enters sleep mode while MemBrain is busy with the corresponding background operation.

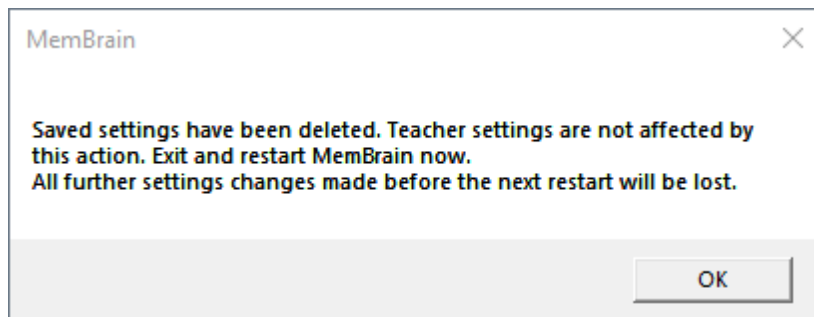
Certainly, MemBrain will remember these settings when it exits, so you only have to adjust them once.

Reset MemBrain to default settings

In cas you want to reset all application settings of MemBrain to defaults select <File><Reset to default settings...> from MemBrain's main menu. The following request will appear:



Selecting <Yes> will delete MemBrain's settings file. The application has to be manually restarted. All additional to settings before the restart will be lost / not saved:



Grouping

Grouping is a feature of clustering neurons and their connected links hierarchically in order to improve editing and comprehensibility of the neural net architecture.

The following chapters explain the related topics:

- [Grouping Elements](#)
- [Ungrouping Elements](#)
- [Selecting Group Members](#)
- [Changing Group Properties](#)

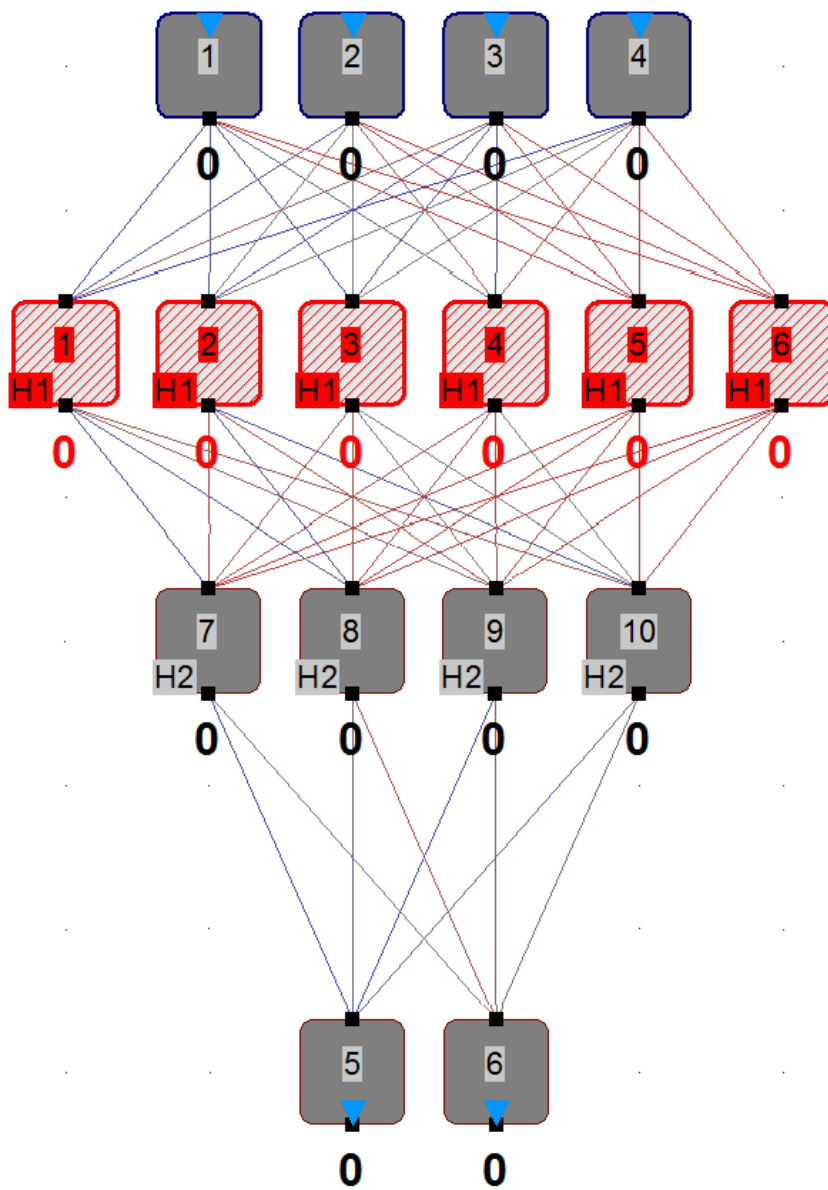
Grouping Elements

In MemBrain neurons can be hierarchically grouped in order to build collapsable regions of the net. These groups can even include other groups in turn.

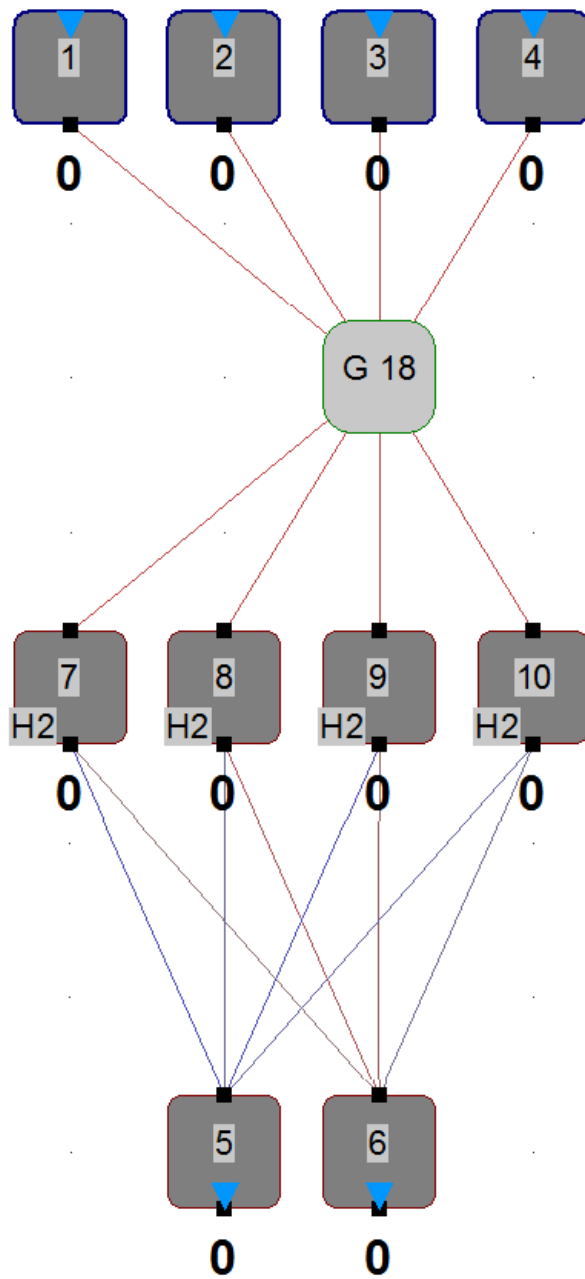
Grouping related actions are performed via one of the following ways:

- Context menu commands when right-clicking on a selected neuron or group
- <Edit> menu, sub section <Grouping>

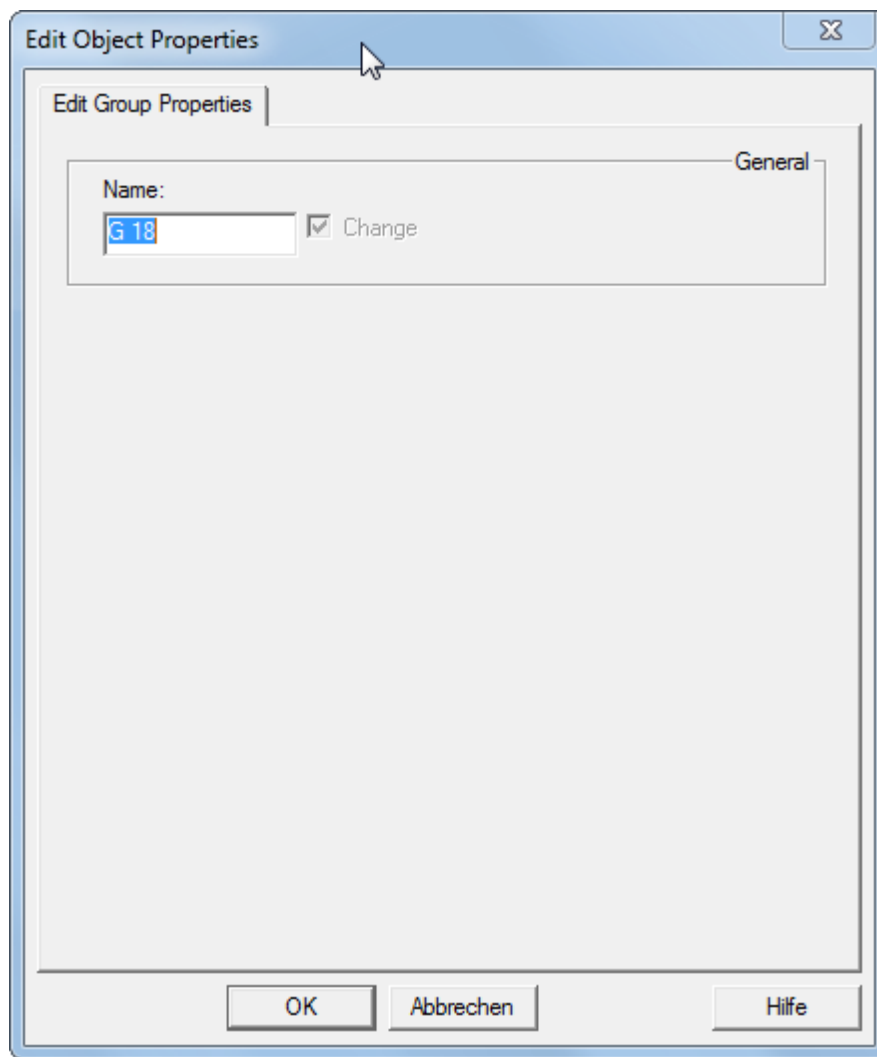
Consider the following example:



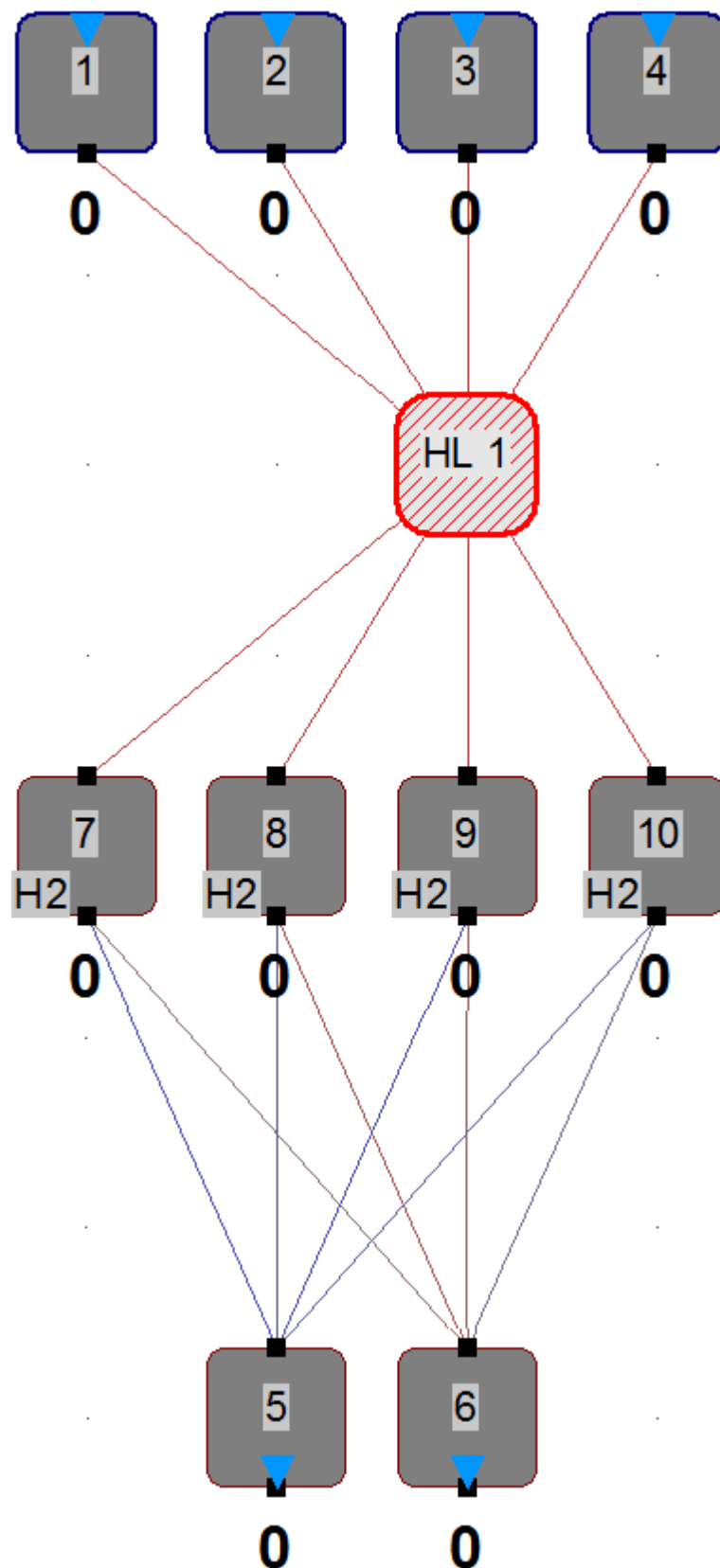
Right clicking on one of the selected neurons and selecting 'Group Element(s)' results in the following:



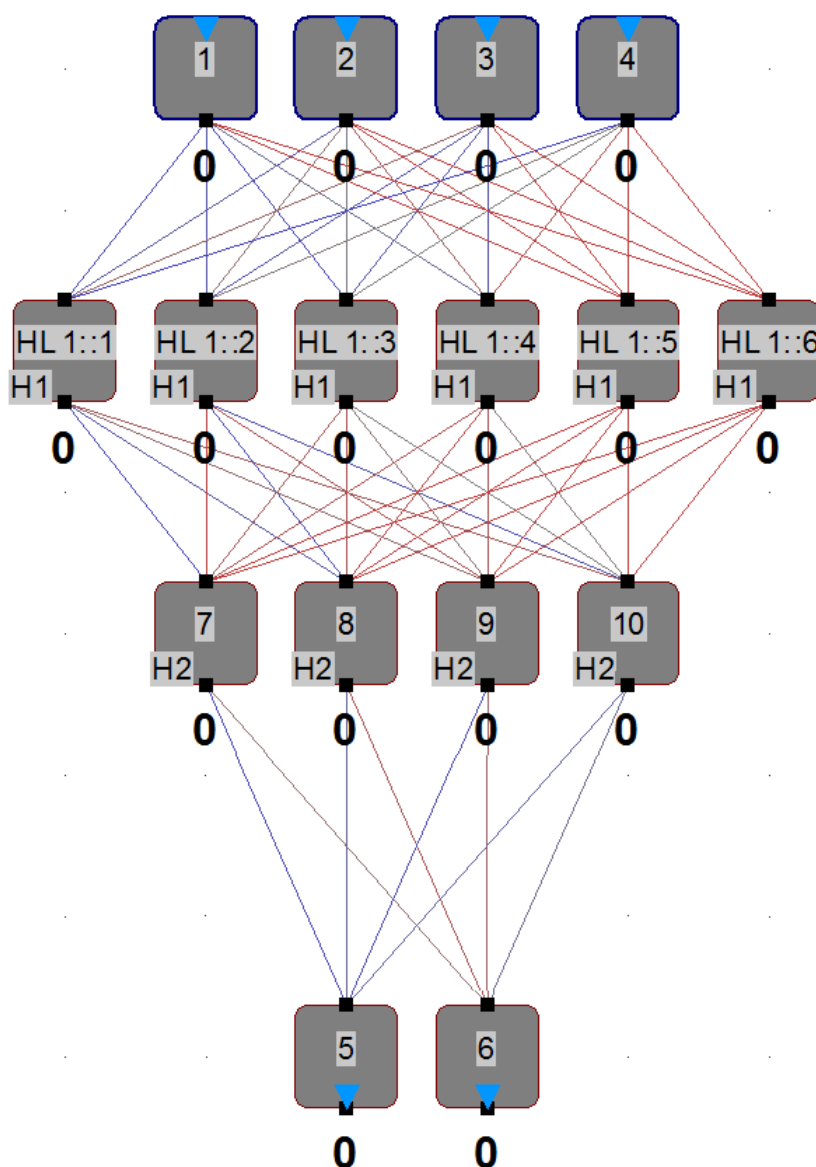
Note that the selected neurons have been collapsed into the newly created group named 'G 18'. This is just a default name generated by MemBrain internally. You can change the name of the group if you want by double-clicking on the group:



E.g. change the name to 'HL 1' which may stand for 'Hidden Layer 1':



If you right-click on the group HL 1 and select 'Ucollapse Group(s)' you will see the following:



Note that the names of the neurons within the group are preceded by the scope identifier 'HL 1::' to indicate that the neurons belong to the group HL 1.

You can collapse the group again by right-clicking on one of the grouped neurons and select 'Collapse Group(s)'.

Adding elements to an existing group

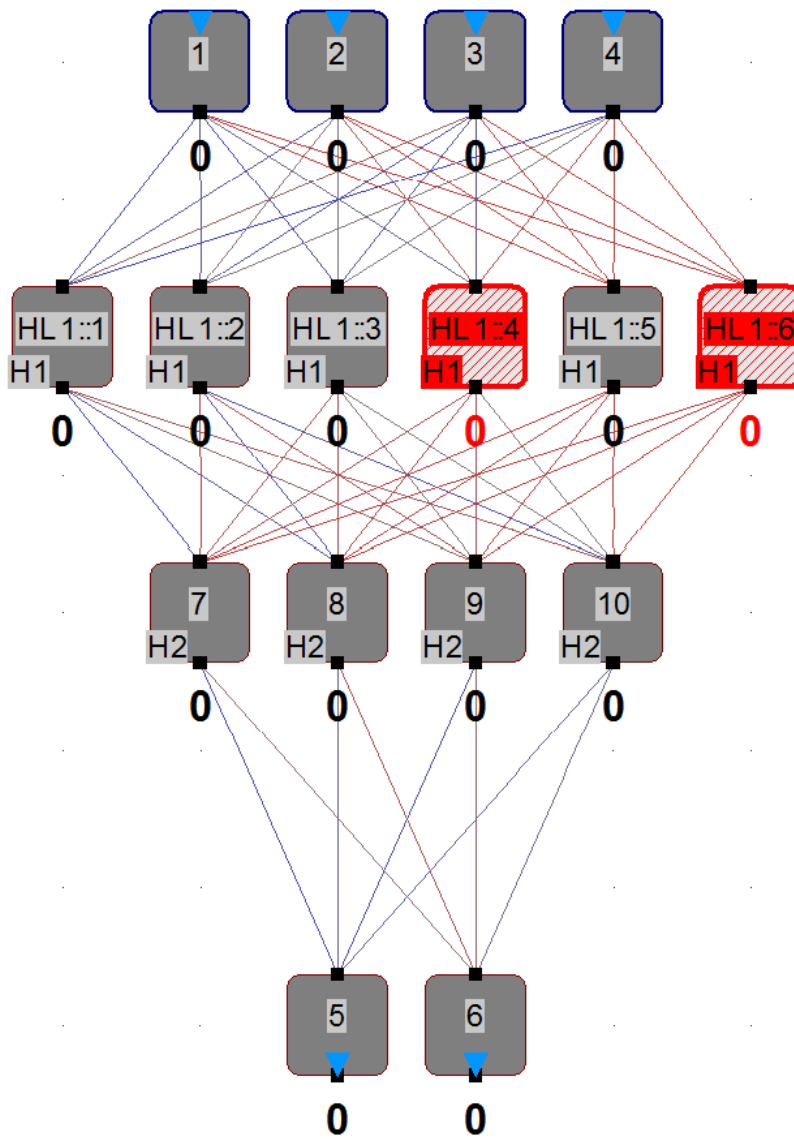
If you want to add neurons or groups to an already existing group do the following:

- [Extra Select](#) the target group you want to add to. Ensure that this is the only Extra Selected element in the net.
- Select the elements you want to add to the Extra Selected group
- Execute <Edit><Grouping><Add to Extra Selected Group> or right-click on one of the selected elements and choose the corresponding entry from the context menu.

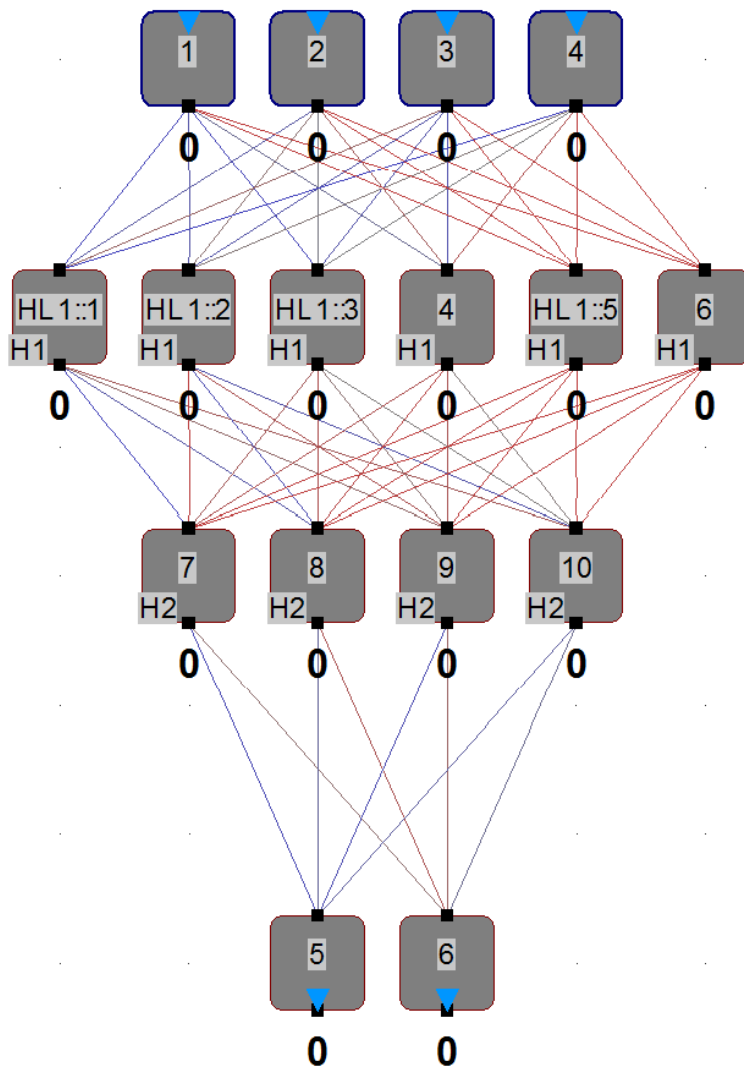
Ungrouping Elements

Elements can be removed from groups by right-clicking on them and selecting 'Ungroup Element(s)'.

Consider the following example:



Right-clicking on one of the selected neurons and then clicking <Ungroup Element(s)> results in the following:



Note that the neurons 4 and 6 now have been removed from the group 'HL 1'.

Selecting Group Members

Grouping can be used to select all members of a group:

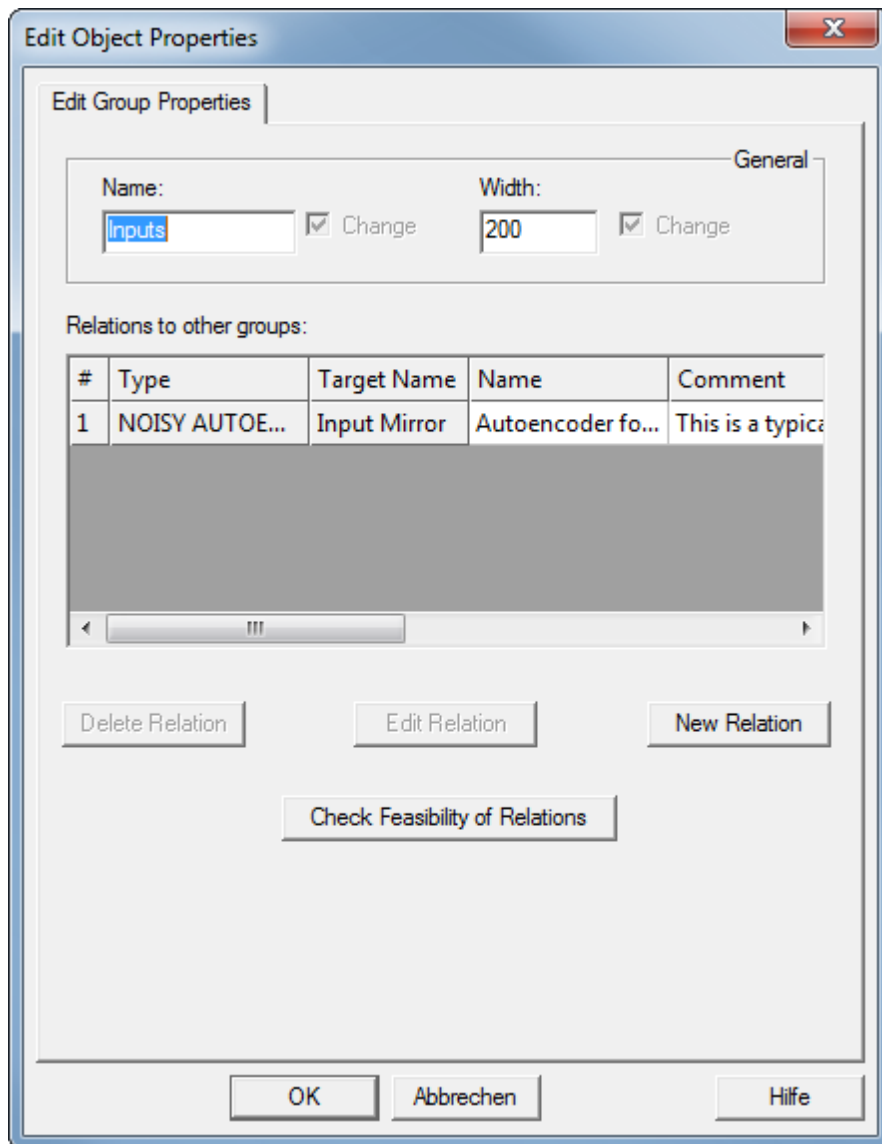
Right click on a neuron or group which is member of a group. Then select <Select Group Members>. All members of the group are selected.

Note that this also works with members of multiple different groups at the same time.

Changing Group Properties

Group properties can be edited just like neurons or links by double-clicking on a collapsed group. Another possibility is to right-click on a neuron which is part of a group and then select <Edit Owning Group(s)...> from the context menu.

The following dialog appears (not that the displayed content will vary depending on your net):



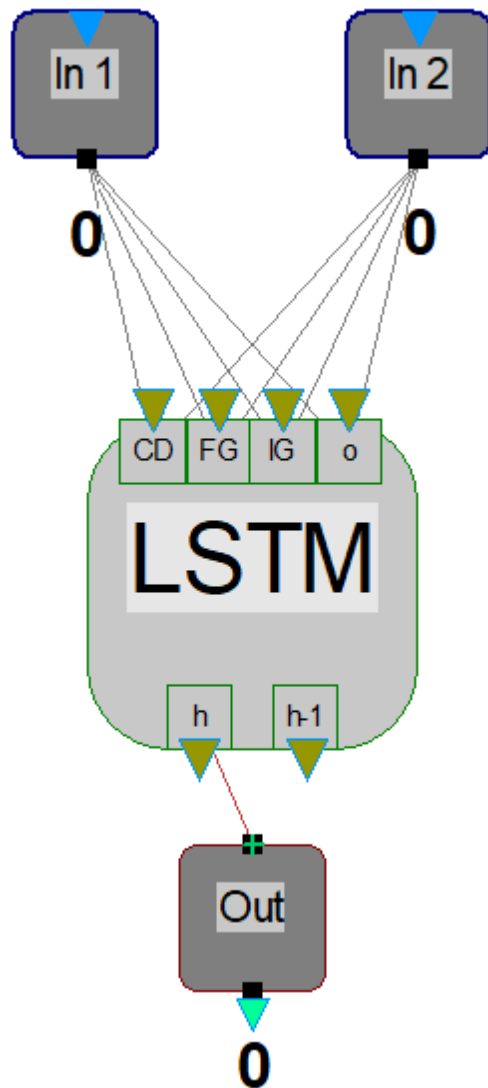
In the top area of the dialog the name of the group(s) and the width used to display the group(s) can be changed.

The bottom area of the dialog is only enabled if you are editing a single group and if there are more than one group defined in the net. In this case it is possible to define and edit relations between the edited group and other groups in the net. This feature allows to define sub nets within the net which can then be trained separately using different approaches.

See [here](#) for more information on defining and editing group relations and what can be achieved by them.

Using Proxy Ports

In order to make input or output ports of grouped neurons available to the outside of their owning group you can use so called Proxy Ports. These are connection points of neurons inside the group which are located on the icon of the collapsed group. The following shows an example:

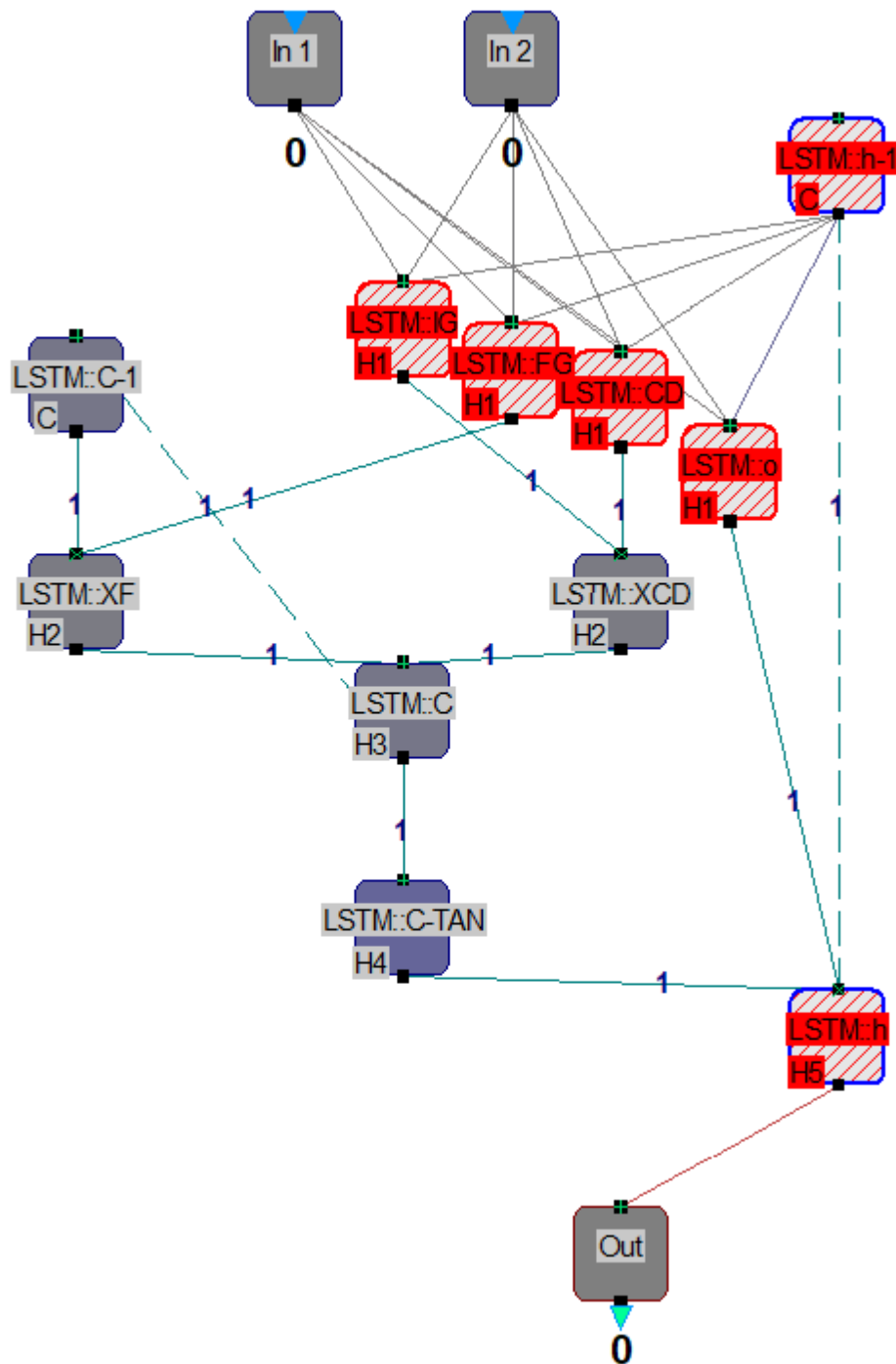


The collapsed group in the example may represent an LSTM (Long Short Term Memory) cell with a hidden dimension of 1 like presented below (uncollapsed group).

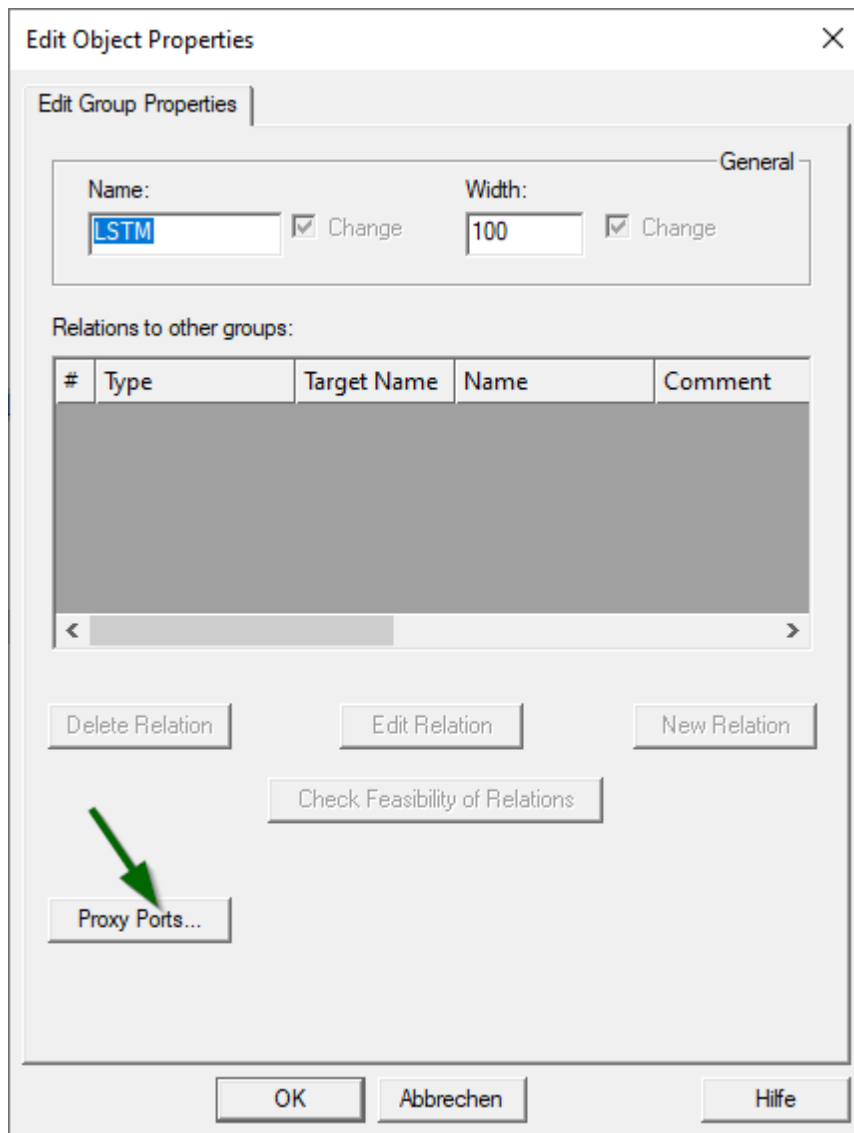
The details on the functionality of the example shall not be discussed here. What is important to show here is that four input ports and 2 output ports of neurons inside the group are visible and available for connection on the collapsed group icon:

The corresponding neurons have been selected in the picture below for illustration purpose

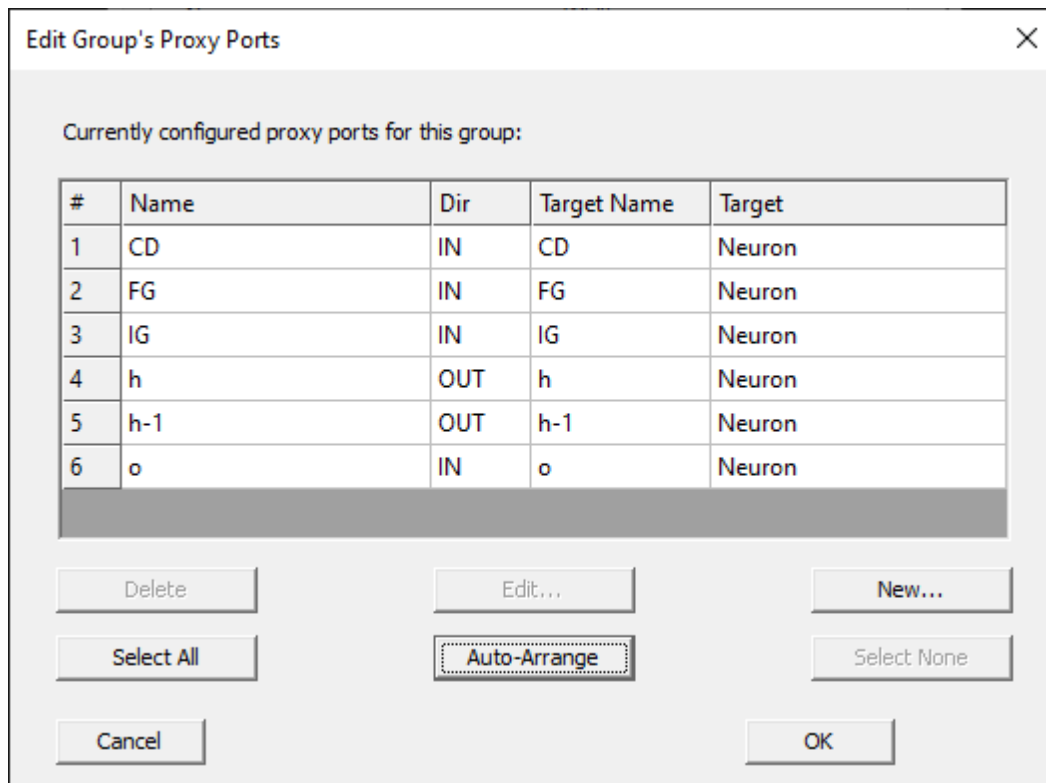
- 4 Input ports of neurons **CD**, **FG**, **IG** and **o**
- 2 output ports of the neurons **h** and **h-1**



In order to review the proxy port assignment of the group, edit the group (by either double-clicking on the collapsed group icon or by selecting "Edit owning group(s)" from the context menu (right click) of any of the neurons which are part of the group. In the "Edit Object Properties" dialog that pops up, click on button <Proxy Ports...>

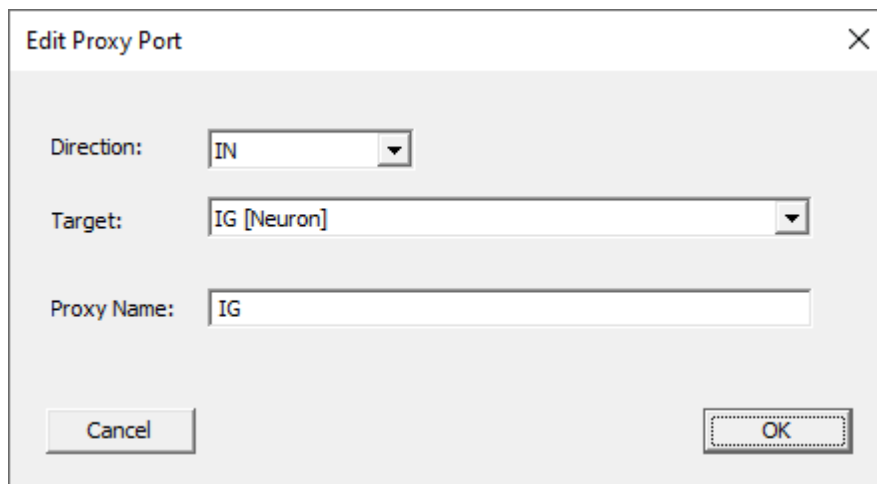


The following dialog appears which shows a list of all proxy ports of the group. As we can see, we can review the name, direction, the name of the target object of the port and the type of the target object. Target objects can either be neurons or groups. In case the target is a group then the proxy port actually refers to a proxy port of an inner group.



The proxy ports can be selected in the list and then either edited or deleted using the buttons below the list. Also, new proxy ports can be added.

For instance, when clicking on the <Edit...> button while the list entry 3 ("IG") is selected the following dialog appears which allows to edit this proxy port:

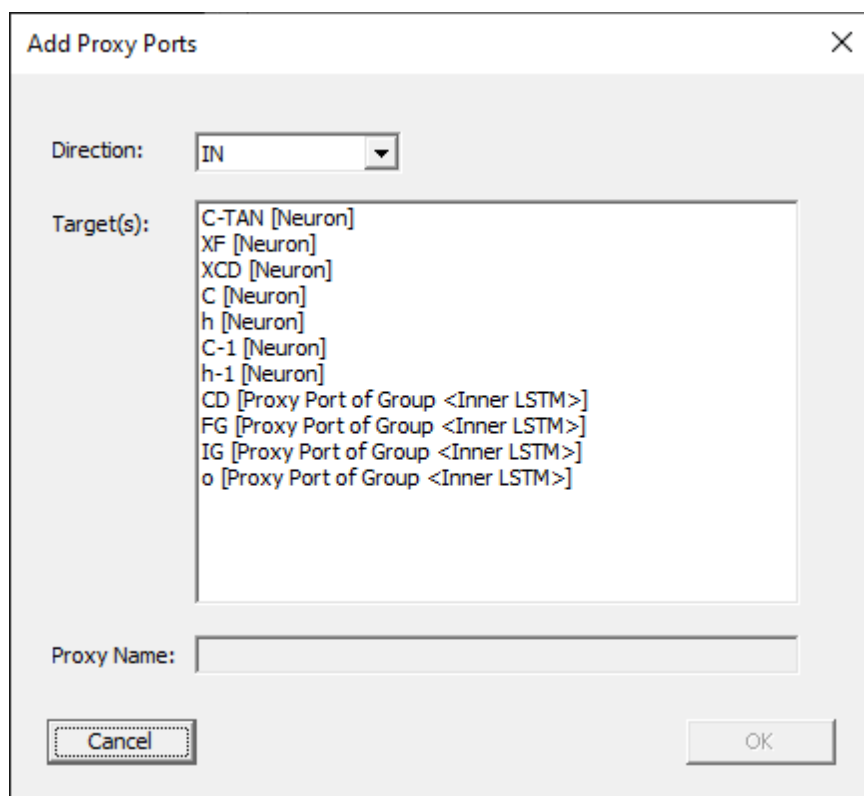


Direction, Target and name of the proxy port can be edited/selected.

The button <Auto-Arrange> performs two different actions simultaneously:

- It sorts all proxy ports in the list alphabetically
- It auto-arranges the proxy port icons on the group's shape in the same order as in the list.

Adding proxy ports is possible via the <New...> button which opens the following dialog

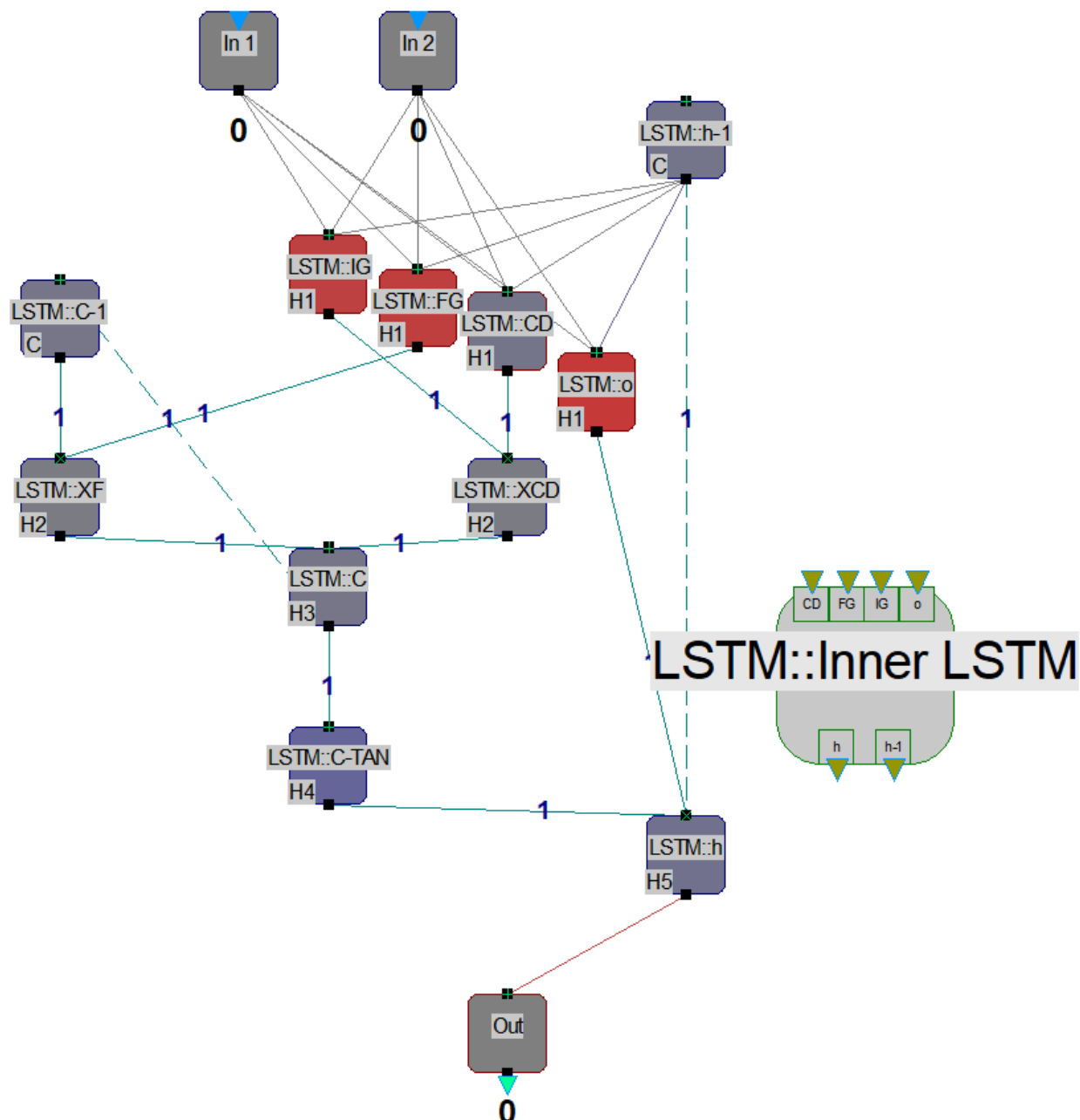


The list automatically is filled with all the available proxy port targets inside the group. Switching between a choice of input or output ports is possible via the <Direction> drop down list box in the top of the dialog. The list allows to perform multi selection:

- Hold the <CTRL> key down during selection: Add or remove items from the selection
- Hold the <SHIFT> key down during selection: Select a range of list entries

Clicking <OK> when one or more list entries are selected will add proxy ports for the selected targets.

Note that for creation of the above list another - inner - group with proxy ports has been added to the above example in order to demonstrate the possibility to connect proxy ports to proxy ports of inner groups. The uncollapsed group <LSTM> for the above example looks like this:



Selecting and connecting proxy ports

Proxy ports can be selected via the following optional ways:

- Right click on a proxy port
- Draw a selection rectangle around one or more proxy ports [just like when selecting other objects](#)

When holding down the <CTRL> key while clicking on a proxy port with the right mouse button, the proxy port will be added/removed from the current selection (toggle action).

Note that proxy ports can also be [Extra Selected](#) in order to make use of [MemBrain's advanced connection features](#).

Neural Networks

Generally in MemBrain there is no 'invalid' neural net. Every net can be simulated no matter of how many neurons it consists of or of what types these neurons are. You don't even need any links in MemBrain to build a valid network!

If you want to use your net in conjunction with input or output data sets that are managed in so called [Lessons](#) then the following requirements apply to your net:

- All input neurons of the net (if any) have to have different names
- All output neurons of the net (if any) have to have different names
- If you want to use a supervised teacher then you need to have at least one output neuron in the net.

There may be as many neurons or links in the net as desired and the neurons can be freely placed in the drawing area. Generally, the geometric position of a neuron on the screen will not influence the behaviour of the net.

In order to get some architectural information about your net you may perform the [Net Analysis](#) command which will give you details of what your net consists of, which logical layers could be identified etc. See [here](#) for more information about the net analysis.

Net Analysis

Before MemBrain can perform mathematical operations on a net, such like calculating the net's output or teaching the net, an analysis has to be performed on the net. This is because of the following.

- The [logical layer structure](#) of the net has to be determined in order to derive rules on how to calculate the net output and how to teach the net.
- If Lesson data shall be used with the net (e.g. during teaching) It has to be ensured that all of the input and output neurons have different names. Additionally teachers that use a supervising learning algorithm need at least one output neuron for teaching.

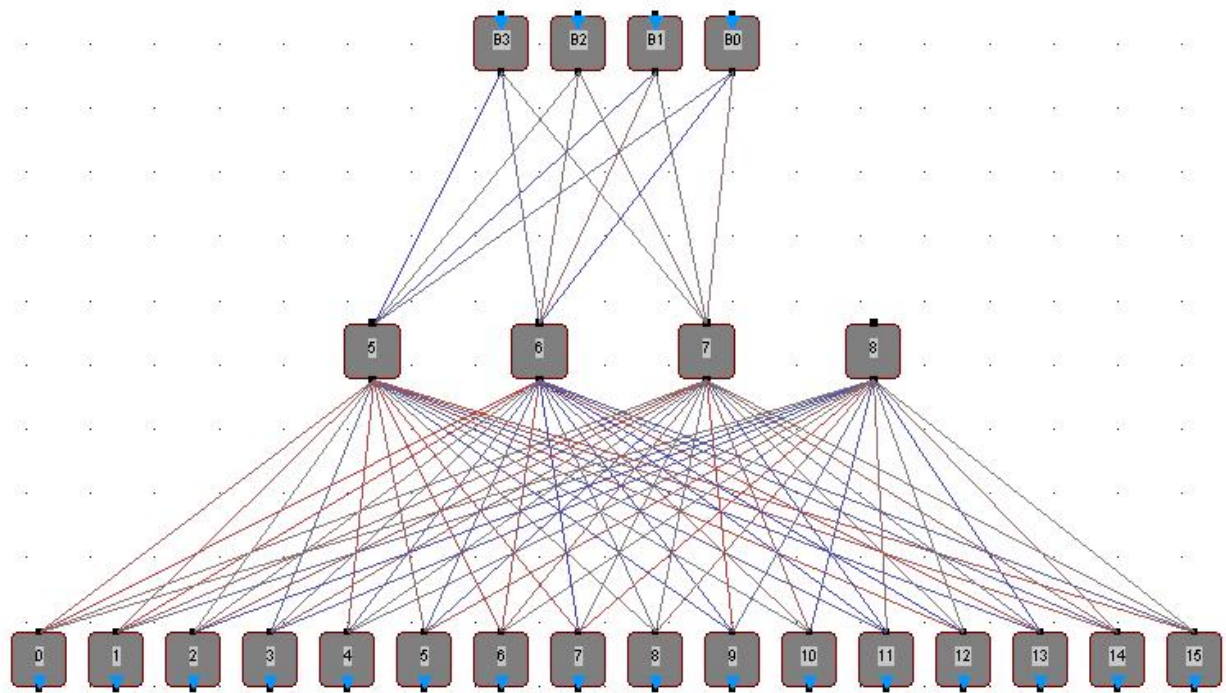
The net analysis is run automatically when you try to perform an action on a net that requires the analysis results. If there are no errors you might not even notice this to happen. MemBrain remembers if the analysis on a net has already been successfully performed and repeats it only if necessary, e.g. when the net architecture has changed. If the analysis results in errors in conjunction with the desired operation MemBrain will inform you about that showing a message box and advise you to use the Net Analysis command to get detailed information about the findings.

You can also run the analysis as a stand alone feature by selecting <Net><Analyse Net> from the main menu whenever you want to get more information about your net.

Architectural Integrity

One thing the net analysis of MemBrain does is to check the net for architectural anomalies. These anomalies do not represent error conditions as in MemBrain every possible net is also a valid net. But the net analysis reports these findings as warnings to give the user a hint on where to take an eye on.

For example if you run the analysis on the following net



you will get the result:

Net Analysis Result

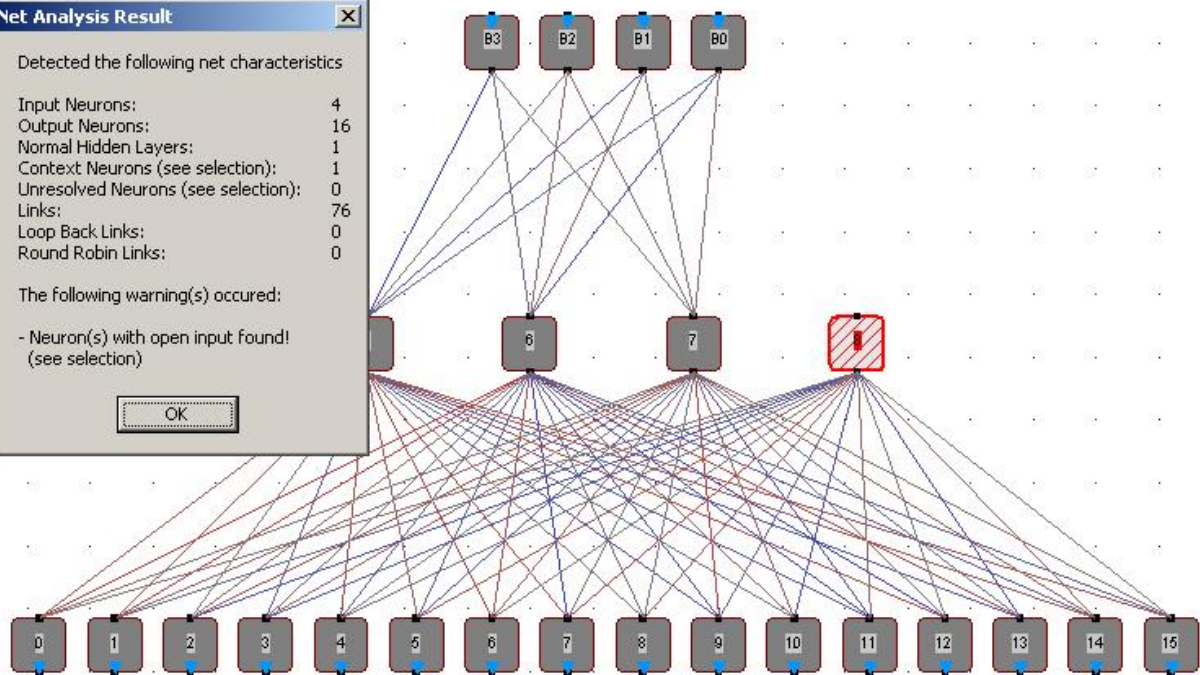
Detected the following net characteristics

| | |
|-------------------------------------|----|
| Input Neurons: | 4 |
| Output Neurons: | 16 |
| Normal Hidden Layers: | 1 |
| Context Neurons (see selection): | 1 |
| Unresolved Neurons (see selection): | 0 |
| Links: | 76 |
| Loop Back Links: | 0 |
| Round Robin Links: | 0 |

The following warning(s) occurred:

- Neuron(s) with open input found! (see selection)

OK



Indicating the warning that the neuron with the name '8' has an open input.

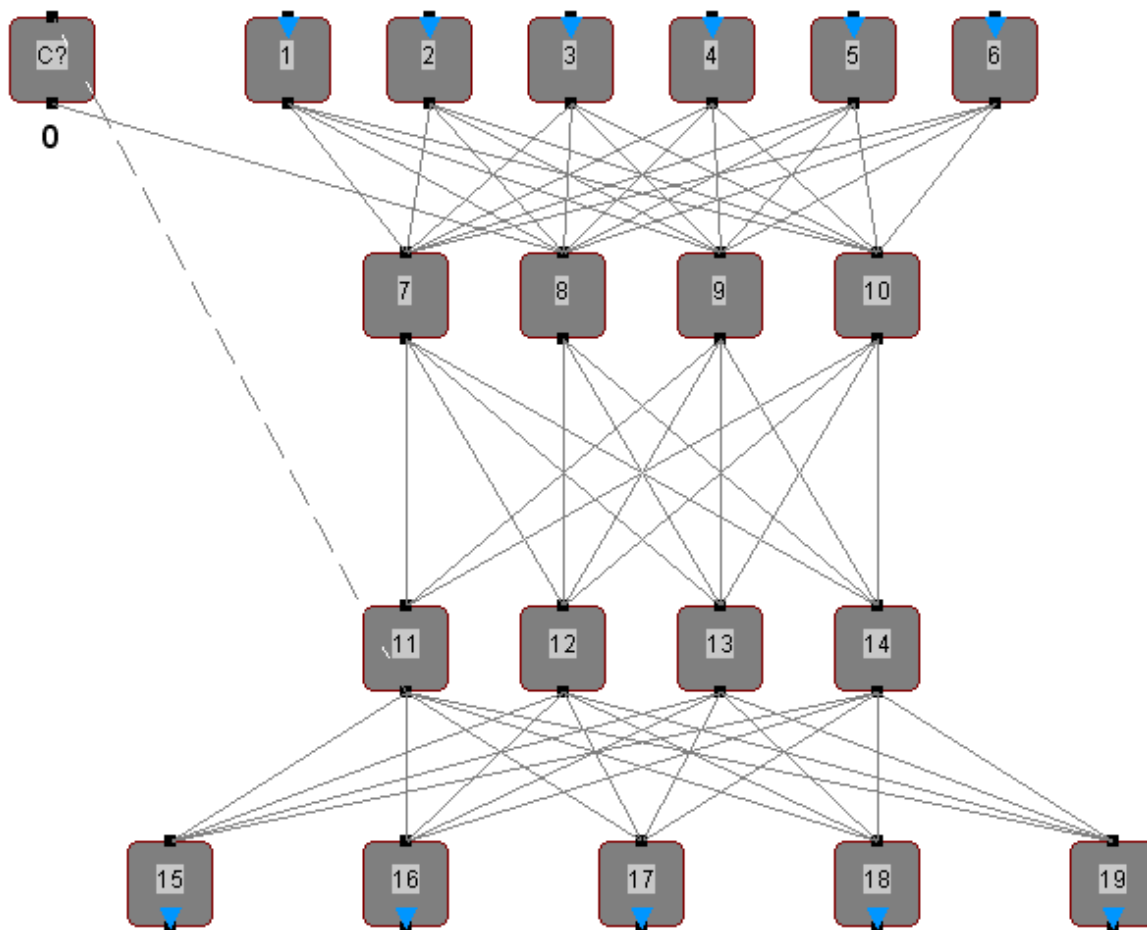
Note that the analysis automatically selects the items of interest so that potential errors can be quickly identified.

In this case the neuron 8 is also identified as a context neuron because its only functionality is to feed back its previous state into the net when the net is simulated as it has got no input connections. Context neurons are neurons that feed back previous activation states into the net.

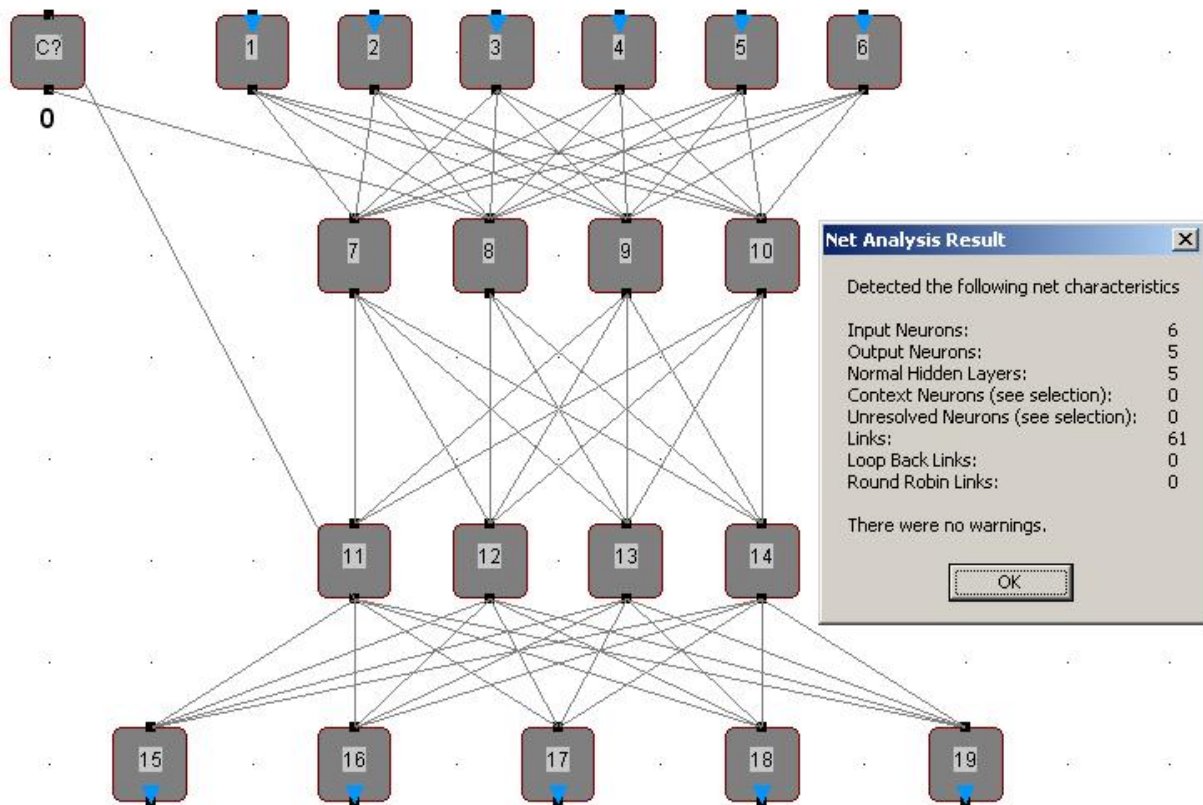
Layer Analysis

The net analysis of MemBrain also determines of how many logical layers the net consists, what type the links in the net are and if there are context neurons contained in the net. To get an idea of what MemBrain does when analyzing a neural net take a look at the following example.

This is a net as it may have be drawn initially. Notice the neuron named 'C?'. It is intended to be a context neuron that feeds back previous output states into the net (in this case the former output of neuron 11). Initially, the link to 'C?' is drawn in a dashed style as it goes upwards (see also [here](#)).



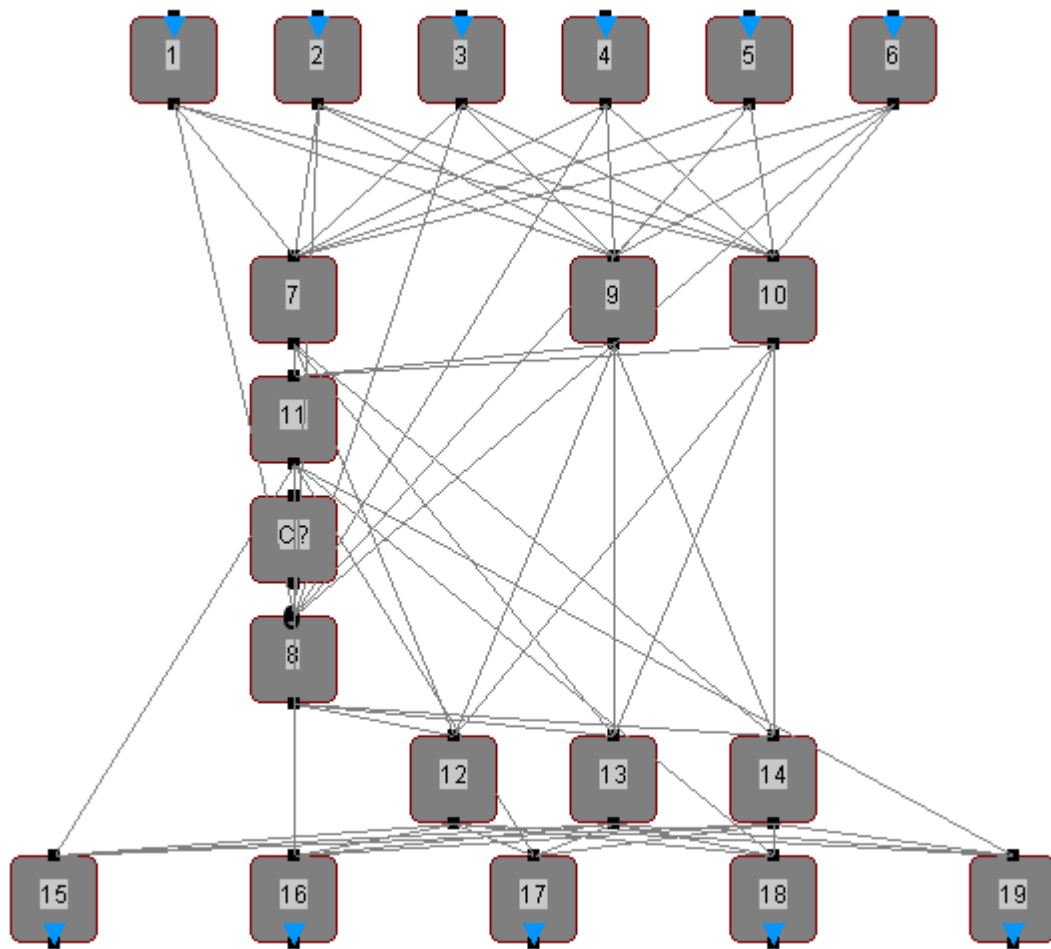
When we run the net analysis (menu command <Net><Analyse Net>) we get the following result displayed by MemBrain.



Note that the input link to 'C?' is not dashed anymore and the result summary tells us that there are 5 normal hidden layers and no context neurons. How can this be?

Here comes the explanation:

When MemBrain performs its net analysis it does not check for screen positions of the neurons to build the logical layers of the net. In this case MemBrain has managed to get a layer solution for the net that doesn't need context neurons. You can see that its possible to re-arrange the neurons on the screen so that no loop back links are there anymore:

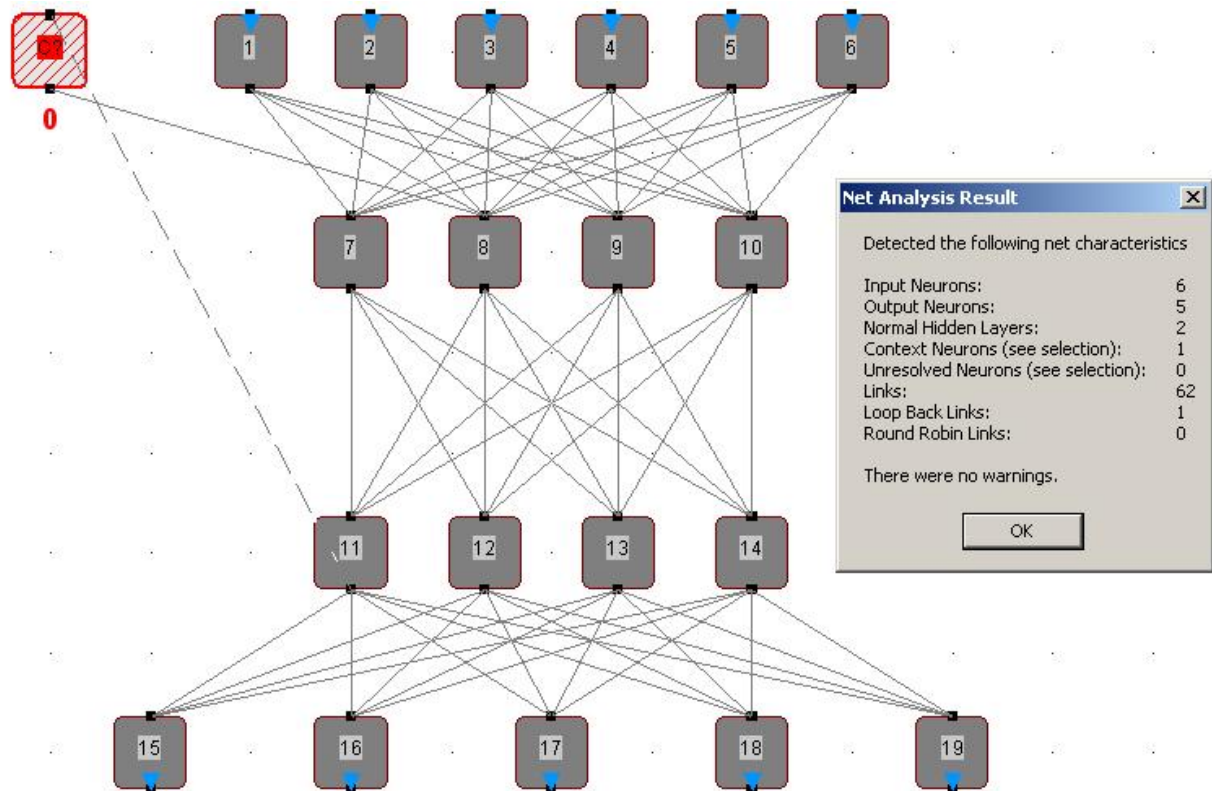


As now easily visible this results in 5 hidden layers of the net, but the net is still a simple feed forward net. There just **are** not loop backs.

Note: If you are unsure about the layers detected within a net you can also tell MemBrain to [display the layer information](#) with every neuron.

When we take a closer look at the first net we drew then we realize that there is no link going from the output of neuron 8 to the input of neuron 11. Actually this is the reason why there is no real loop back in the net.

Lets verify this by adding the missing link and then re-run the analysis:



Now 'C?' is stated to be a context neuron by the net analysis and its input link is drawn dashed as this is a real loop back link. You now can even move the context neuron below neuron 8 or even 11 and the link styles will not change anymore as this link is now determined to be analysed. The number of normal hidden layers is 2 as expected.

Note: Sometimes there are several possible solutions when the net is analyzed. If the result found by MemBrain turns out not to be what is actually desired, then the analysis can be influenced by tagging specific neurons as preferred candidates for becoming context neurons. This is done via editing the [neuron properties](#).

Displaying Layer Information

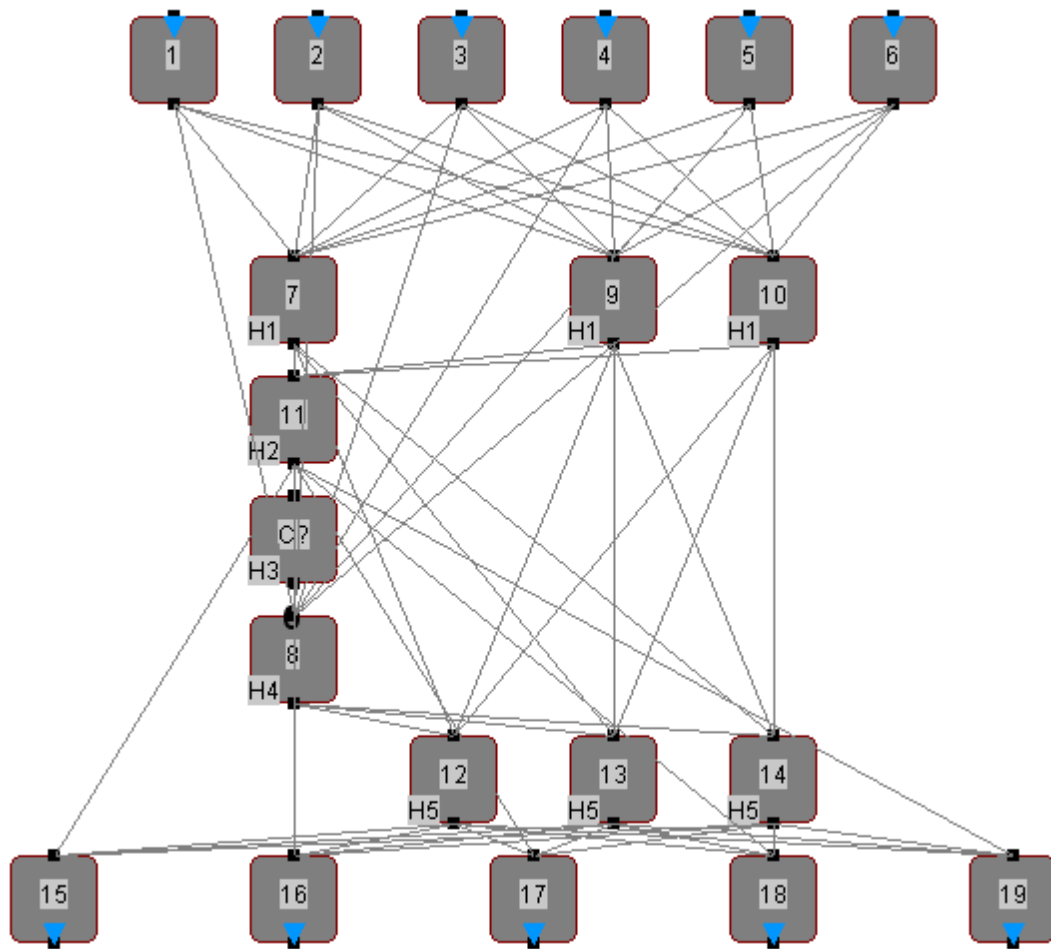
The Layer information for every neuron can be displayed and hidden by use of the menu item <View><Show Layer Info>.

The layer information is shown in the bottom left corner of every neuron as either

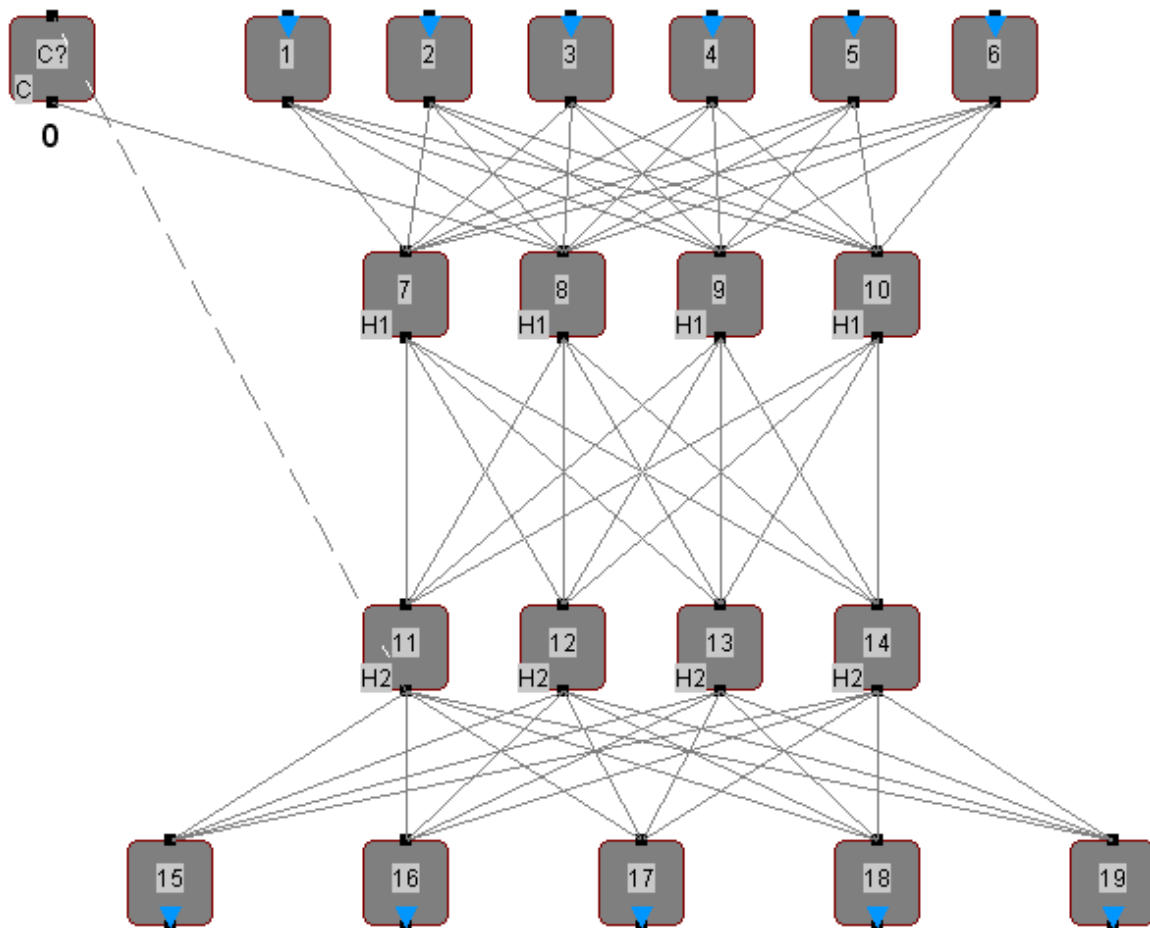
- "Hx", indicating hidden layer No. x, starting with x = 1 for the first hidden layer
- "C", indicating that the neuron belongs to the context layer (feedback layer)
- "?" when the layer cannot be identified or the [Net Analysis](#) has not been run yet.

The following examples show the nets already discussed together with the [Layer Analysis](#).

Net 1:



Net 2:



Calculating the Output

When MemBrain calculates the activations and output signals of the neural net (this procedure is referred to as "Think" in MemBrain) then it does that according to the following layer order.

- Input Layer
- Context Layer (because context neurons activations and output signals derive from the state of the last think step).
- All Hidden Layers beginning with the first layer following the input layer.
- Output Layer.
- Unresolved neurons layer

This procedure is referred to in MemBrain as one "Think Step". According to this terminology you will also find a command in the main menu called <Net> <Think Step> tied to the corresponding tool bar symbol




The new state of the net after the think step will be displayed immediately on the screen.

You can also switch MemBrain to a mode called "[Auto Think](#)" so that the net is permanently re-calculated by repeating the above procedure over and over again. Note that the display is only updated during 'Auto

Think' mode if the option <View><Update View during Think> is enabled!

In the status bar MemBrain always displays the current number of Think Steps performed. You can reset that counter by selecting <Net><Reset Think Step Counter> from the main menu. The think step counter is automatically reset if you start a new 'Auto Think' procedure.

Resetting the Net

If you choose <Net><Reset Net> this will reset all activations of all neurons in the net and also all resident activation spikes on all links to 0. Alternatively, you can use the corresponding tool bar button .

This function is needed when you construct a time variant net, i.e. a net where the next net output depends on the previous internal state of the net. This is the case if your net contains one or more of the following.

- Loopback links
- Links with logical length > 1
- Hidden or output neurons with an Activation Sustain Factor > 0

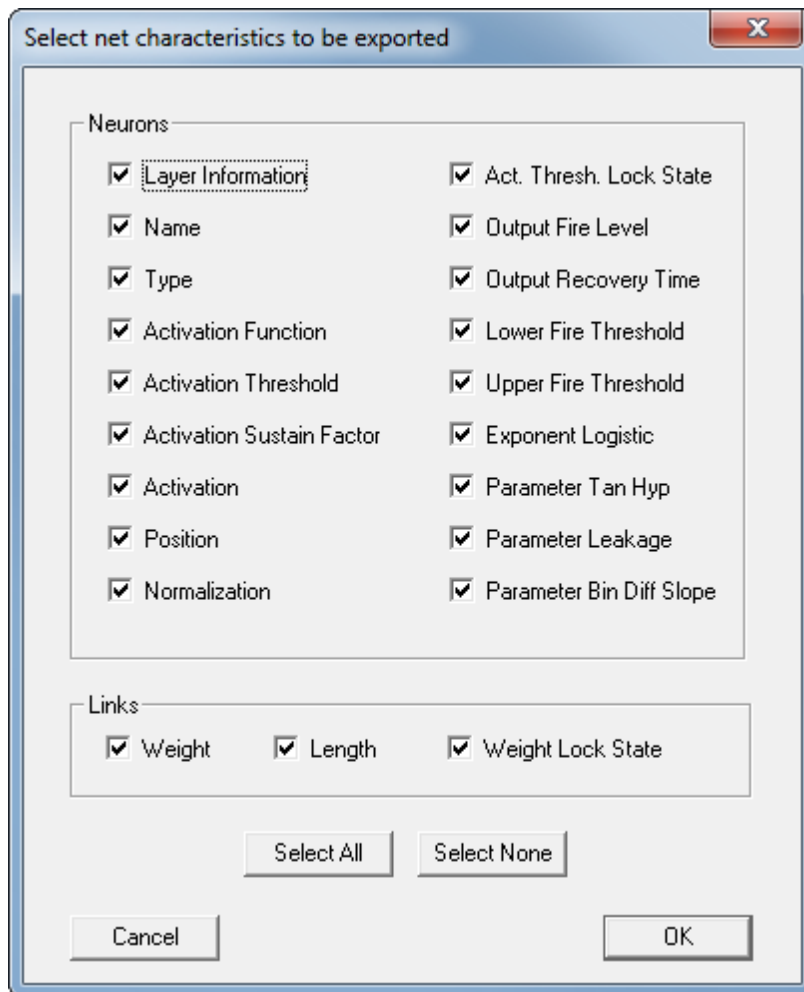
You will need this function then to reset your net to an initial, known state.

Exporting a Net

With MemBrain you have the possibility to export a net list of the currently opened neural net via a csv (comma separated values) file. This file can be used to import a neural net created and possibly trained with MemBrain into some other application software.

You can select the level of details you want to incorporate into the export file.

To export a net select <Edit><Export...>. The following dialog will pop up.



This dialog lets you select the properties of neurons and links you want to have exported into the file. Note the two buttons in the middle which let you select or deselect all properties in one shot.

When you have made your choice click OK. You will be prompted to select a location and file name to store the exported file to.

For an explanation of the net export format see [here](#).

Net CSV File

When you [export](#) a net then MemBrain creates some kind of Sectioned CSV File for that purpose. This file consists of sections of data that are identified by several key words each of them included in a start and an end tag '<' and '>', respectively. This chapter explains how the exported data is formatted.

The following example shows the contents of a csv file created by the MemBrain Neural Network Export function. Explanations to the contents of the file are added in *green italics* here. Note that the exact format of the file depends on the properties you have [selected](#) for export.

Also, the separators used to separate data items and decimal points in the numbers will depend on your computer system's country settings. The following example file has been generated under German country settings.

Information about MemBrain version, build date and file type.

MemBrain, Version XX.XX
(<Month> <Day> <Year>)

Sectioned CSV File

The Start key word for the net.

[<NET START>]

Additional information on the file content

[<INFO HEADER>]

This file represents a MemBrain neural net.

Information about the format used for exporting neurons.

This is some kind of 'headline' to explain the data contained in the [<NEURONS<] section which follows next.

[<NEURON FORMAT INFO>]

ID;LAYER;NAME

This section contains all neurons in the net. One neuron always is represented by one line in the csv file.

[<NEURONS>]

1;l;In1

2;l;In2

3;o;Out

Information about the format used for exporting links.

This is some kind of 'headline' to explain the data contained in the [<LINKS<] section which follows next.

[<LINK FORMAT INFO>]

SOURCE_ID;TARGET_ID;WEIGHT

This section contains all links in the net. One link always is represented by one line in the csv file.

[<LINKS>]

1;3;0,263535

2;3;0,178995

End key word.

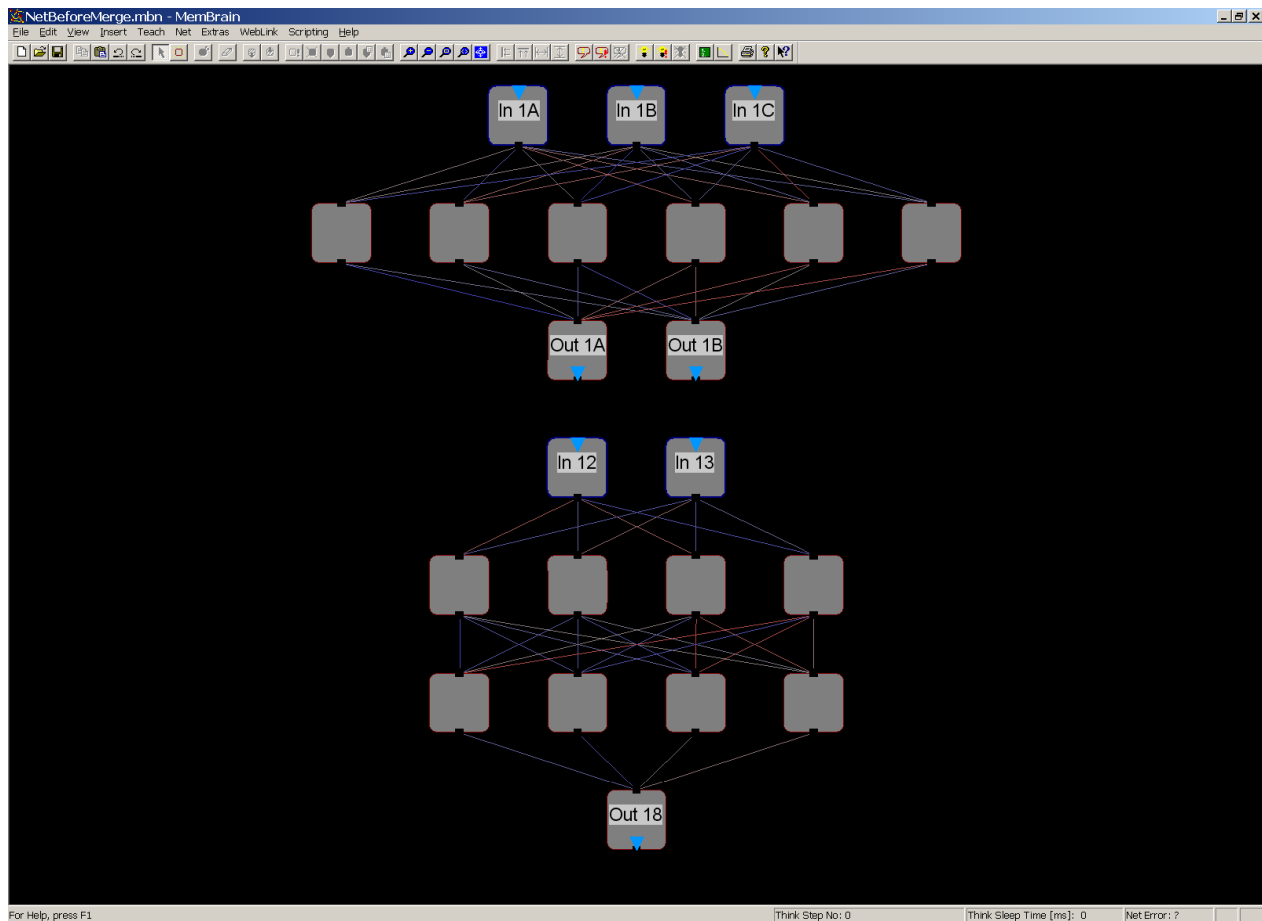
[<END>]

Merging Nets

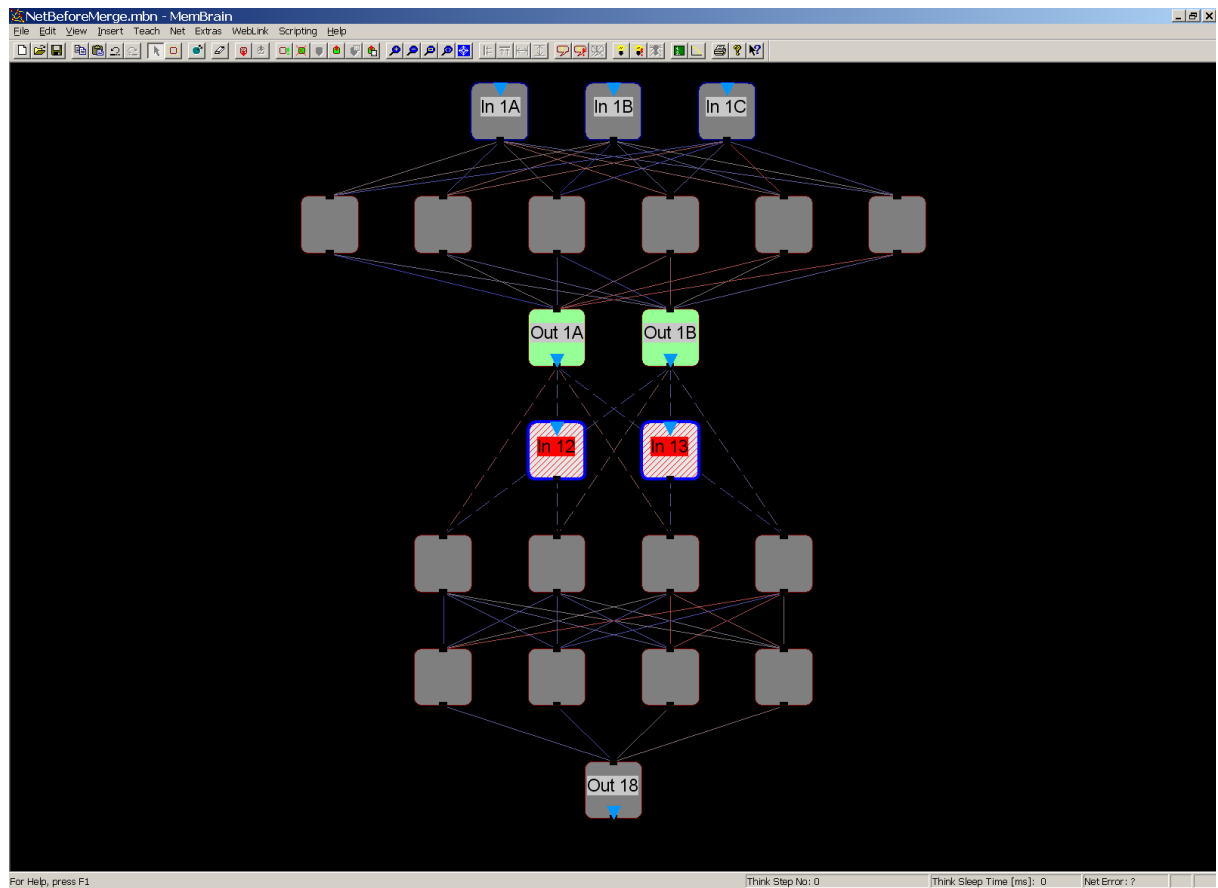
This chapter describes how it is possible to merge different trained sub-nets together into one single bigger net.

1. Open the first net in MemBrain
2. Add the second net to the first net using the menu command <File><Add Net to Current Net...>

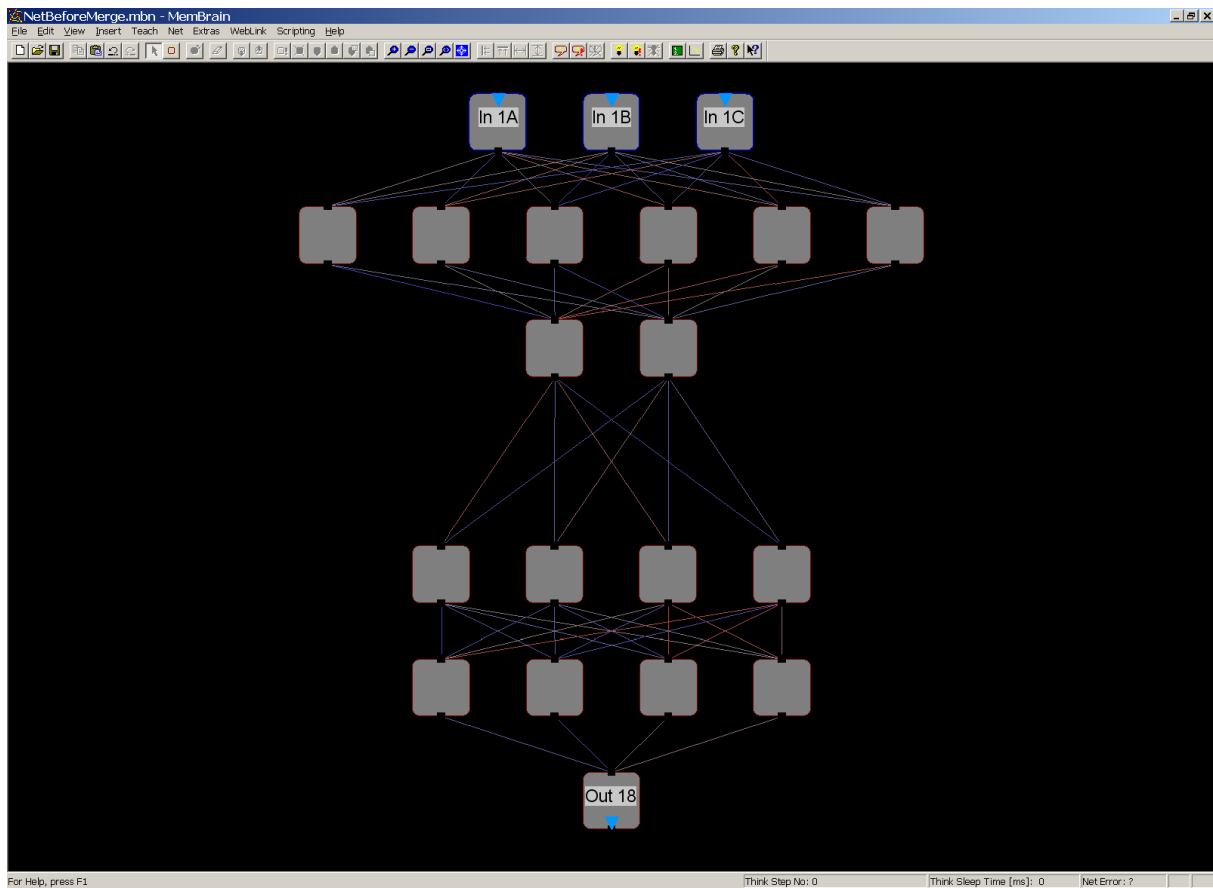
You now have both the nets inside MemBrain, this might look as in the following example.



3. Now you have to ensure that the former outputs of the upper net build the new inputs for the lower net. You can do this by re-assigning the already trained output links of the input neurons of the lower net to the output neurons of the upper net:
[Extra Select](#) the output neurons of the upper net. Select the input neurons of the lower net. Then choose **<Extras><Re-assign output links from Selection TO Extra Selection>**. See [here](#) for more information on link re-assignment.
4. The net now looks as following.



If you now delete the old input neurons of the lower net (which are not connected anymore) and change the type of the old output neurons of the upper net to <HIDDEN> then you have merged the two nets into one single net:



Note that the re-assigned links may appear dashed. That is, because a new net analysis has not been run. You can [do that manually](#) if you want to or leave it up to MemBrain which will do it if required during further operations.

Managing I/O Data

Input/Output data in MemBrain is organized in so called "Lessons".

A **Lesson** is a collection of input/output data sets (so called **Patterns**) for a specific net. A Pattern is one set of input/output data for a net that belongs together i.e. one value for every input neuron of the net and one value for every output neuron.

If a neural net has been trained with every Pattern of one lesson one time this is called one "Epoch" in the neural net literature. In MemBrain this is also referred to as one 'Lesson Run' or also as one 'Teach Step'.

In MemBrain you can administer Lessons using the [Lesson Editor](#). With that tool you can create template Lessons out of an existing net or rename your net input/output neurons according to your lesson. You also can capture and edit complete Patterns from the current net with one mouse click. Certainly lessons can also be saved to or loaded from files using the Lesson Editor.

One more important feature of the Lesson Editor is the capability of recording data to Lessons. E.g. if you want to validate a net trained with MemBrain you can use the Lesson Editor to perform a Think Step on every Pattern of a Lesson (option 'Think On Lesson') and record the results from the net into another Lesson. You can later on export the recorded Lesson to csv and evaluate it using standard spread-sheet programs for example.

Data for input neurons is always mandatory for a lesson while the data for the output neurons can be deactivated for a lesson. This is useful if you want to train your net using non-supervised learning algorithms ('teachers'). In this case you don't need data for the output neurons as this data is not known during the network training.

If you want to create, view or modify your lessons with external software you have also the possibility to export or import Lessons from a CSV text file. For more information on the available file formats click [here](#) and [here](#).

You can also [normalize](#) your I/O data to the appropriate ranges separately for every input or output neuron.

The Lesson Editor

When working with neural nets you'll have to frequently deal with creating, administering saving and loading input and output data. Also data has to be recorded for example to validate the net.

For this purpose the Lesson Editor has been added to MemBrain. The Lesson Editor can be opened by choosing <Teach><Lesson Editor> from the main menu or by simply clicking



on the tool bar.

The lesson editor opens as following

Lesson Editor: unnamed

Lesson Files MemBrain CSV Files Raw CSV Files Image Files Extras

▲ Number of Lessons: 1
 ▲ Currently Edited (Training) Lesson: 1
 ▲ Net Error Lesson: 1

☐ Set Manually

Input data:

| In 1 | In 2 | In 3 | In 4 | In 5 | In 6 | In 7 | In 8 | In 9 | In 10 |
|------|------|------|------|------|------|------|------|------|-------|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Pattern Name: Pattern No: of 1

☒ Output Data:

| Out 1 | Out 2 | Out 3 | Out 4 | Out 5 | Out 6 | Out 7 | Out 8 | Out 9 | Out 10 |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|--------|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Sync With Net

Number of Inputs:

Number of Outputs:

Data to Net

Data from Net

Record Lesson

☐ Record one pattern every

1 Think Step

To Lesson No. 1

☒ Activations ☐ Outputs

Name of Lesson:

The Lesson Editor can administer as many [lessons](#) as you want. When you open the lesson editor after MemBrain has been started it always has already loaded one lesson that is equipped with one empty data pattern (all data is zero).

The Lesson Editor always shows one [Pattern](#) (if the currently selected lesson is not empty), that is, one set of data for the input neurons and one corresponding data set for the output neurons of the net. Both rows of data are displayed in one grid line each, every column of the grid represents one input and one output neuron, respectively.

If you have a net opened in the main window and the input/output neurons do not have duplicate names then the input and output data column names will be the same as in the net when the lesson editor is opened the first time. If you first open the Lesson Editor and do *not* have a net loaded or if the IO names cannot be retrieved from the net because there are duplicate input or output names (for which you will get an error message) then there will be no input and output neuron names defined in the lesson editor. You can synchronize the data column names with the net later on using the command buttons on the Lesson Editor, however.

The number of lessons currently administered by the Lesson Editor is pointed out in the top left of the dialog ("Number of Lessons: ###"). You can add new empty lessons by increasing this number with the up/down arrows aside. If you decrease the number the lessons are deleted again. If you try to delete a lesson that is

not empty (has at least one pattern) you will be prompted by the Lesson Editor before deletion takes place. You cannot delete lesson number "1". The lesson editor always has at least one lesson opened.

The number of the currently edited lesson is shown in the upper right corner of the Lesson Editor ("Currently Edited (Training) Lesson: ###"). Modify this number by the arrows aside. The displayed data will update according to the lesson that has the focus. All operations accessed via the Lesson Editor are performed on this lesson with the one exception of recording data during think which is explained further down in this chapter. It is very important to note that the 'Currently Edited' lesson always is the lesson that is used to train your net!

The Lesson Editor is a non modal dialog i.e. it can be used in parallel to the main view. You do not have to close the Lesson Editor when working on the net. Closing the Lesson Editor does not even affect its entered data. It will show up again when re-opening the Lesson Editor.

Just be sure to save your Lessons before to close the MemBrain main window (i.e. exit MemBrain) as there is no warning implemented that ensures that Lesson Editor related data is saved when closing MemBrain! Unsaved data of the current lesson will be discarded without notice! To save your data use <Lesson Files><Save Current Lesson> or <Lesson Files><Save Current Lesson As...> or <Lesson Files><Save All Lessons> from the Lesson Editor's menu.

You can perform the following functions with the Lesson Editor:

- Add/Delete lessons
Use the up/down arrows in the upper left corner of the dialog to add/delete whole lessons. The overall number of administered lessons is displayed beneath the arrows.
- Select the active (Training) lesson
With the up/down arrows in the upper middle area the Lesson Editor you can select which of the administered lessons is the currently active one. All operations (except for recording patterns) are effective on the active lesson only.
Also this is the lesson that is used by Teachers during training.
- Select the Net Error lesson
In the upper right corner of the lesson editor you can select which of the loaded lessons shall be the basis for the teachers to calculate their net error during teaching. Normally, this lesson is the same as the active lesson, i.e. when teaching you see the net error based on the training lesson. However, you can manually set the Net Error lesson to be a different one by selecting the check box below the corresponding arrows and then changing the value by clicking the up/down arrows.
Thus, it is possible to use one lesson for training (the active lesson) and calculate the net error on basis of another lesson which usually is a lesson that contains untrained validation data. This allows to validate a net 'On The Fly' during teaching.
- Select the active pattern:
The currently active pattern of the lesson can be changed by the arrow buttons on the right hand side of the dialog. You can also use the mouse wheel to scroll the active pattern upwards or downwards.
Additionally, you can enter a specific pattern number to jump to directly in the corresponding edit field.
- Edit Pattern Name:
You can assign a unique name to every pattern of a lesson. Simply type the name into the edit field between the input and the output data row.
- Add Pattern Comment:
You can add a comment to every pattern of a lesson. For the moment the comments are not further used in MemBrain but will be included in the [Lesson Export File](#), where they may be useful for your future reference to the data.
- Edit Lesson Name:

You can assign a unique name to the lesson. Simply type the name into the edit field at the lower right corner of the Lesson Editor.

- Add Lesson Comment:

You can add a comment to the lesson. For the moment the comments are not further used in MemBrain but will be included in the [Lesson Export File](#) where they may be useful for your future reference to the data.

- Simple Data Editing:

You can edit the currently active input/output pattern by simply clicking into the edit fields of the input or output data grid and changing the values.

- Delete Pattern:

To permanently delete the current pattern of the loaded lesson click on the button <Delete Pattern>.

Note: There will be no request to confirm the action and there generally is no Undo/Redo option in the Lesson Editor. So be careful when using this button!

- Delete All Patterns:

To permanently delete all patterns of the loaded lesson click on the button <Delete All Patterns>. You will be requested to confirm this action. Note that it cannot be undone!

- Add new Pattern:

In order to add a new pattern to the lesson click on the button <New Pattern>. A new pattern filled with all 0's will be created and added at the tail of the current lesson. The new pattern will get the currently displayed (i.e. active) one no matter which pattern has been displayed beforehand.

- Synchronize Lesson with the Net:

In order to use a lesson in conjunction with a neural net the lesson and the net must be synchronized. This means that the input and output names and count must be the same at the lesson and the net. To achieve this there are several possibilities:

1. *Manually define I/O count and names:*

You can manually change the number of inputs and outputs of the current lesson by typing the values into the corresponding edit fields of the Lesson Editor. Press the <Apply> buttons beneath the edit fields to let your changes have effect. Note: Changing these numbers will affect all patterns of the lesson! You will be warned when reducing one of these numbers as this results in loss of data (deleted columns).

Also you can change the input and output names of the lesson. Click on <Edit/Lock Names> and the names will get editable. After you have finished changing the names click <Edit/Lock Names> again as this protects the names from being unintentionally changed.

2. *Names from Net:*

When you click on the button <Names from Net> then the lesson will be automatically formatted according to the currently loaded net in MemBrain's main window. You will be requested to confirm the action because it may result in loss of data if the net has less input or output neurons than the current lesson in the editor has. Typically this is an action that is performed when creating new lessons to an existing net or when the I/O names of the net do not fit the ones of the lesson but the number and column order does.

Important Note: When the I/O names are synchronized these will be retrieved from the net and assigned to the lesson columns from left to right to achieve the same geometric arrangement in the lesson and the net.

The algorithm is the following: Take the top most input neuron and assign its name to the left most input column of the lesson. Then search for all right hand neighbors of this input neuron and assign their names to the next right hand neighbors in the lesson editor. If there are no right hand neighbors in the net anymore then search for the next bottom input neuron neighbor in the net and continue with adding its name to the next right neighbor

column in the lesson. This algorithm has been chosen to automatically transfer matrix formatted input neuron fields (e.g. representing a picture that consists of rows and columns) in a useful manner to a lesson.

When all input neurons have been synchronized the same procedure will be performed for the output neurons.

3. *Names to Net:*

This action transfers the I/O names from the current lesson to the I/O neurons of the net that is opened in the main window. The function will only succeed if the numbers of input and output neurons are the same for both the lesson and the net. You will get an error message if this is not the case. The algorithm used for identifying the neurons that are assigned the names of the lesson columns is corresponding to the one described in the previous section.

- Think on Current Pattern Input:

When you click on the button <Think on Input> then the input data of the currently active pattern will be applied to the input neurons of the opened net and the net will perform one [Think Step](#). You will get an error message if the action fails. You can also use this button during [Auto Think](#) mode.

- Think on Next Pattern Input:

When you click on the button <Think on Next Input> then the input data of the next pattern following the active one in the lesson will be applied to the input neurons of the opened net and the net will perform one [Think Step](#).

You will get an error message if the action fails. You can also use this button during [Auto Think](#) mode. Use the mouse wheel to scroll up and down through the lesson and perform a think step on the pattern selected by the scrolling action. To do this press and hold the <CTRL> key on the keyboard while scrolling.

- Think on Lesson

When you click on the button <Think on Lesson> then all input data patterns of the current lesson will be applied to the input neurons of the opened net and the net will perform one [Think Step](#) on each of the patterns. The procedure starts with the first pattern in the Lesson and ends with the last pattern. You will get an error message if the action fails.

- Think on L. + Winner

When you click on the button <Think on L. + Winner> then the behaviour is the same as with button <Think on Lesson>, except that the winner output neuron for each pattern is renamed according to the corresponding pattern name. You will get an error message if the action fails.

- Take Pattern from Net:

If you click the button <Pattern from Net> then the current pattern will be overwritten by the current activations of the input respectively output neurons of the net. You will be requested to confirm the action.

- Add New Pattern from Net:

Clicking the button <New Pattern from Net> creates a new pattern with the current activations of the input respectively output neurons of the net. You will not be requested to confirm the action as there will be no loss of data. The new pattern will be placed at the end of the lesson and will be displayed as the active pattern.

This feature can be used to very efficiently create new patterns in combination with the [Quick Activation](#) option. Use the Quick Activation to set the inputs and desired outputs of your net and then click on the button <New Pattern from Net> on the Lesson Editor which will add the pattern to the lesson.

- Record Lesson:

You can automatically record new patterns of a lesson by checking the box on the lower right side of the lesson editor. If it is activated the lesson editor will add one pattern to the adjusted lesson every time the specified number of think steps has been performed. You can adjust the pattern recording

interval by clicking on the corresponding up/down arrows. Also, the lesson to which the new patterns shall be added can be adjusted.

With this feature it is easy to gain validation data on your net: Create a new lesson and adjust the recording to that lesson. Then set the active lesson to the one that provides your test patterns. Let the net think about all patterns in the test lesson by the button <Think on Lesson>. The response of your net together with the applied input data will be recorded to the new lesson after every think step.

Note that it doesn't matter if the Think Steps are performed in [Auto Think](#) mode, by [manually triggering](#) them or during [teaching](#) the net.

If you record a lesson in [Auto Think](#) or [Auto Teach](#) mode the lesson editor will not be actualized with the new data until you stop the simulation/teach process again.

With the two radio buttons <Activations> and <Outputs> you can select which data of the output neurons to be recorded to the lessons. The default value is <Activations>. This causes the activation values of the output neurons to be recorded. If you change the selection to <Outputs> then the output values (fire level) of the output neurons are recorded instead of the activation values. This is useful if you use the setting '1' instead of 'Activation' as the output fire level of the output neurons:

You can adjust the fire thresholds of the output neurons to a value which matches your decision criteria when the neuron shall be considered as active and when as inactive. Normally you would set both lower and upper fire threshold to the same value e.g. 0.5. This would cause the neuron to output a value of '1' when the activation is higher than 0.5 and else a 0. These values will be recorded to the lesson if you select the mentioned option <Outputs>.

Note: It is important to mention that it is possible to also record data to the currently active lesson. In most cases this is not desired, so be sure to select a lesson different from the active lesson for the recording. Also, ensure that recording is disabled again once you have finished it in order to prevent inadvertent recording actions during further operations!

- Save the Lesson:
Use <Lesson Files><Save Current Lesson> respectively <Lesson Files><Save Current Lesson As...> from the Lesson Editor's menu in order to save the currently edited lesson in the MemBrain internal lesson format.
- Save all Lessons
To save all lessons in one shot choose <Lesson Files><Save all Lessons> from the Lesson Editor's menu.
- Load a Lesson:
Lessons can be loaded into the Lesson Editor using <Lesson Files><Load Current Lesson...> from the Lesson Editor's menu. Note that this will erase all previously entered data of the currently active lesson! If you want to add patterns from another lesson to the currently edited lesson then use <Lesson Files><Append to Current Lesson...> instead.
- Append a Lesson to the current one
You can append all Patterns in a MemBrain Lesson file to the currently edited Lesson, i.e. combine two smaller Lessons into one. Use <Lesson Files><Append Lesson to Current Lesson...> from the Lesson Editor's menu.
- Export Lesson as CSV file:
The currently edited lesson can be exported as MemBrain CSV formatted data for further processing in other applications like spread sheet programs for example. Use <MemBrain CSV Files><Export Current Lesson...> from the Lesson Editor's menu. The export format is detailed [here](#).
- Import Lesson from CSV file:
The contents of the currently edited lesson can be imported from MemBrain CSV formatted data created

or modified with other applications. Use <MemBrain CSV Files><Import Current Lesson...> from the Lesson Editor's menu.

The required format is detailed [here](#).

- Export Lesson as Raw CSV file:

The currently edited lesson can be exported as Raw CSV formatted data for further processing in other applications like spread sheet programs for example. Use <Raw CSV Files><Export Current Lesson (Raw CSV)...> from the Lesson Editor's menu. The export format is detailed [here](#).

- Import Lesson from Raw CSV file:

The contents of the currently edited lesson can be imported from Raw CSV formatted data created or modified with other applications. Use <Raw CSV Files><Import Current Lesson (Raw CSV)...> from the Lesson Editor's menu. The required format is detailed [here](#).

- Export Lesson Inputs as Raw CSV file:

The input data of the currently edited lesson can be exported as Raw CSV formatted data for further processing in other applications like spread sheet programs for example. Use <Raw CSV Files><Export Current Lesson Inputs (Raw CSV)...> from the Lesson Editor's menu. The export format is detailed [here](#).

- Import Lesson Inputs from Raw CSV file:

The input data of the currently edited lesson can be imported from Raw CSV formatted data created or modified with other applications. Use <Raw CSV Files><Import Current Lesson Inputs (Raw CSV)...> from the Lesson Editor's menu. The required format is detailed [here](#).

- Export Lesson Outputs as Raw CSV file:

The output data of the currently edited lesson can be exported as Raw CSV formatted data for further processing in other applications like spread sheet programs for example. Use <Raw CSV Files><Export Current Lesson Inputs (Raw CSV)...> from the Lesson Editor's menu. The export format is detailed [here](#).

- Import Lesson Outputs from Raw CSV file:

The output data of the currently edited lesson can be imported from Raw CSV formatted data created or modified with other applications. Use <Raw CSV Files><Import Current Lesson Inputs (Raw CSV)...> from the Lesson Editor's menu. The required format is detailed [here](#).

- Split Current Lesson:

Often it is useful to split a Lesson into two parts: One for training and one for validation.

In MemBrain you can perform this split activity automatically through the Lesson Editor: Select <Extras><Split Current Lesson...> and enter the percentage that shall be split from the currently active Lesson. The Patterns which are split off are then moved to a new Lesson that is automatically created. The new Lesson will always be the last Lesson in the Lesson Editor. After the Split action you will notice that the current Lesson now has less patterns than before while the number of Lessons in the Lesson Editor has been increased by one. Use the up arrow on the right hand side of the Lesson Editor to navigate to the new Lesson that has been created. The patterns which are split off to the newly created Lesson are ordered according to their original positions within the source lesson.

When splitting off lesson data you have the option to either select the split patterns by random or let MemBrain split off a certain percentage of the lesson's tail. The latter option should be chosen if the data is used to train/validate a time cariant net: In this case the order of the data patterns must be respected so that random split off is not acceptable. MemBrain will automatically pre-select the appropriate split-off method accorcing to the currently loaded net. However, you can always manually override MemBrain's pre-selection [here](#).

- Work with input data only:

If you want to work with input data only you can deactivate the output data of the lesson:

Remove the check mark beneath the text 'Output Data:' above the row that shows the data for the output neurons. Now the data gets dimmed. The Lesson Editor will now only work with the input data and disrespect the output data. This is useful if you want to train your net using non-supervised learning algorithms ('teachers'). In this case you don't need data for the output neurons as this data is not known during this kind of the network training.

- Read gray scale image files into the inputs of the currently selected pattern of the active lesson.
See [here](#) for details.
- Calculate FFT (Fast Fourier Transform)
See [here](#) for details.

FFT Calculations

The Lesson Editor provides the functionality to calculate FFTs (Fast Fourier Transforms) of your data.

An FFT is a frequency spectrum representation of time series data.

If your time series data provides **N time points** then the resulting FFT will have **(N / 2) + 1 frequency components** where the term 'N / 2' is rounded down to the nearest integer value.

The frequency for every component calculates as :

$$f_i = i / \text{sampleTime}$$

where 'i' is the index of the FFT component starting with $i = 0$. I.e. i ranges from 0 to $N / 2$.

The following options are available in the <Extras> menu of the Lesson Editor for this:

- **Create Absolute Value FFT From Current Lesson Rows**

This option creates a new lesson that includes one pattern (i.e. one row) of FFT data for every pattern (i.e. row) in the currently active lesson.

Use this function if:

- Your time series data is contained in the rows of the current lesson, i.e. when every row in the current lesson represents a time series for which the frequency spectrum shall be calculated.
- If you are not interested in the phase of the FFT frequency components but only need their amplitude.

The newly created lesson will have $N / 2 + 1$ inputs named 'f0' .. 'fx' where $x = N / 2$.

The outputs of the current lesson will be copied to the new lesson without any changes.

- **Create Complex Value FFT From Current Lesson Rows**

This option creates a new lesson that includes one pattern (i.e. one row) of FFT data for every pattern (i.e. row) in the currently active lesson.

Use this function if:

- Your time series data is contained in the rows of the current lesson, i.e. when every row in the current lesson represents a time series for which the frequency spectrum shall be calculated.
- If you are interested in the phase of the FFT frequency components i.e. need a complex value representation of the frequency spectrum with a real and an imaginary part for every frequency.

The newly created lesson will have $2 * (N / 2 + 1)$ inputs named 'f0_R', 'f0_I' .. 'fx_R', 'fx_I' where $x = N / 2$.

f_i_R represents the real value part of the complex frequency component with index 'i'.

f_i_I represents the imaginary value part of the complex frequency component with index 'i'.

The outputs of the current lesson will be copied to the new lesson without any changes.

- **Create Absolute Value FFT From Current Lesson Columns**

This option creates a new lesson that includes one pattern (i.e. one row) of FFT data for every column in the currently active lesson.

Use this function if:

- Your time series data is contained in the columns of the current lesson, i.e. when every column in the current lesson represents a time series for which the frequency spectrum shall be calculated.
- If you are not interested in the phase of the FFT frequency components but only need their amplitude.

The newly created lesson will have $N / 2 + 1$ inputs named 'f0' .. 'fx' where $x = N / 2$.

The newly created lesson will have no outputs.

- **Create Complex Value FFT From Current Lesson Columns**

This option creates a new lesson that includes one pattern (i.e. one row) of FFT data for every pattern column in the currently active lesson.

Use this function if:

- Your time series data is contained in the columns of the current lesson, i.e. when every column in the current lesson represents a time series for which the frequency spectrum shall be calculated.
- If you are interested in the phase of the FFT frequency components i.e. need a complex value representation of the frequency spectrum with a real and an imaginary part for every frequency.

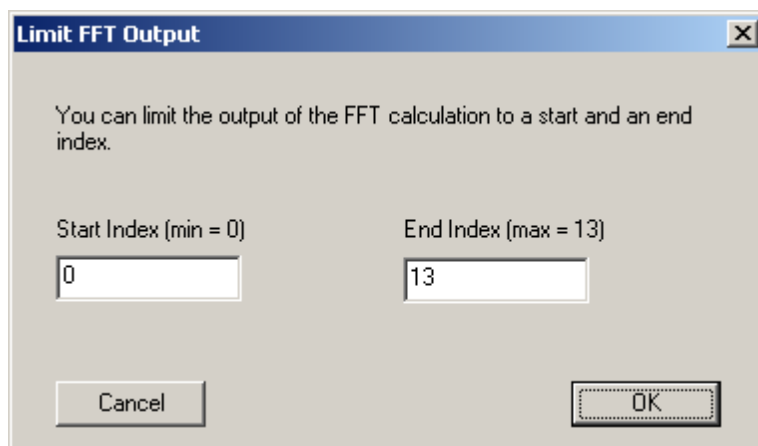
The newly created lesson will have $2 * (N / 2 + 1)$ inputs named 'f0_R', 'f0_I' .. 'fx_R', 'fx_I' where $x = N / 2$.

fi_R represents the real value part of the complex frequency component with index 'i'.

fi_I represents the imaginary value part of the complex frequency component with index 'i'.

The newly created lesson will have no outputs.

Every generated FFT can be limited with respect to the start index and the end index that will be copied to the new lesson. If you select one of the menu items mentioned above then the following dialog will appear.



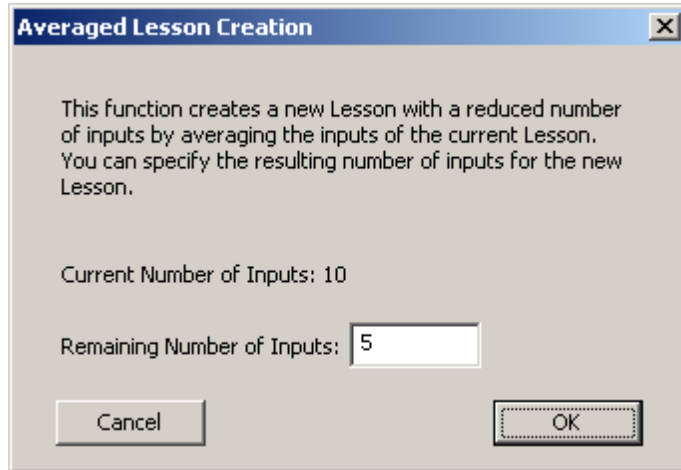
The dialog will prevent you from entering invalid values and will always be initialized with the maximum

number of frequency points possible for the new FFT lesson based on the time series data found in the current lesson.

Averaging Inputs

The Lesson Editor provides a function to automatically the data in the input columns and thus create a new lesson with a reduced number of inputs.

To do so select <Extras><Create Lesson with Averaged Inputs...> from the Lesson Editor's menu. The following dialog will appear:

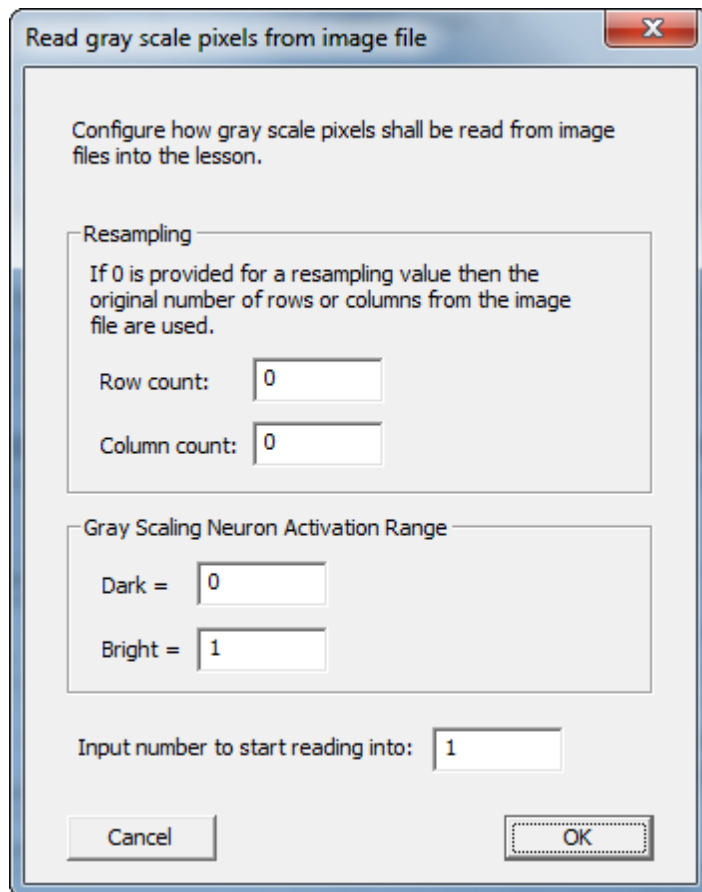


This window allows you to adjust the desired number of remaining inputs in the new lesson. By default the window always shows half the number of the currently available inputs, i.e. the default setting averages every two inputs to for one new input.

The function will create a new lesson in the editor that contains the reduced number of inputs gained by averaging the inputs of the original lesson. The output data of the currently active lesson will be copied to the new lesson automatically.

Reading gray scale pixels from image files

Via the Lesson Editor menu item <Image Files><Read Image File to Current Pattern Inputs...> it is possible to read in bitmaps from a large variety of image file types into the input data array of the currently selected pattern of the active lesson. MemBrain first provides a file selection dialog which allows to select the image file to read from. Then, the following dialog is presented to provide mor eoptions to configure the image read procedure:



Option group "Resampling"

The settings in this group allow to provide a new number of rows and columns to be used to re-sample the image file before the gray scale conversion and reading into the pattern inputs is performed. The default values '0' in these edit fields cause MemBrain to use the number of rows or columns as in the original image file. Typically, one would enter numbers matching the layout of an input neuron pixel matrix here.

Option group "Gray Scaling Neuron Activation Range"

Here, the values for dark (black) and bright (white) pixels can be specified. MemBrain will linearly interpolate between these two values when generating the gray scale pixel values.

Input number to start reading into:

This option allows to start reading into the lesson at a specific input neuron number. This can be useful if the input neuron field contains of multiple pixel arrays (i.e. multiple logical 'images') and only a specific sub section of the input neurons shall be subject to the read action.

Sectioned Lesson CSV File

Sectioned Lesson CSV File

With the [Lesson Editor](#) you have the possibility to export or import lesson data via csv files. This is a useful feature if you want to support data creation by external applications or if you want to further process your lesson data for other purposes like presentation or validation for example.

MemBrain supports two different types of CSV files:

- So called 'Sectioned CSV' file (described in this chapter)
- [Raw csv files](#)

This chapter deals with the Sectioned CSV file format.

The Sectioned CSV file format of MemBrain consists of sections of data that are identified by several key

words each of them included in a start and an end tag '<' and '>' , respectively. This chapter explains how the exported data is formatted and which of the sections are mandatory when importing data from a sectioned CSV file into MemBrain.

The following example shows the contents of a csv file created by the MemBrain Lesson Editor's Export function. Explanations to the contents of the file are added in *green italics* here.

The separators used to separate data items and decimal points in the numbers will depend on your computer system's country settings. The following example file has been generated under German country settings.

Information about MemBrain version, build date and file type.

Not required for import:

MemBrain, Version XX.XX

(<Month> <Day> <Year>)

Sectioned CSV File

There can be any additional comment placed here. This will be ignored by the import functionality.

The Start key word for the lesson. Required for import.

Reading data begins after this key word:

[<LESSON START>]

Optional section. Only for information. Ignored at import:

[<INFO HEADER>]

This file represents a MemBrain Lesson.

Name of the lesson. Optional. Not required for import:

[<LESSON NAME>]

And Gate Complete

Comment for the lesson. Optional. Not required for import:

[<LESSON COMMENT>]

Contains all patterns possible for a simple AND gate

Input list section. Required for import. Lists all input neuron names, separated by the separator character of your country:

[<INPUTS>]

In1;In2

Output list section. Required for import. Lists all output neuron names, separated by the separator character of your country:

[<OUTPUTS>]

Out

Informational purpose only. Ignored at import.

Shows the assignment of the data listed in the section [<PATTERNS>] to the input and output neurons.

[<PATTERN FORMAT INFO>]

In1;In2;Out

Pattern list section. Lists all the patterns of the lesson.

Data of one pattern starts with the first input neuron and continues until the last output neuron.

Every pattern starts with a new line.

Required at import.

Note: A pattern can also span several lines. The length of every pattern is defined by the sections [<INPUTS>] and [<OUTPUTS>].

MemBrain will read in the data only by reading packages of that size,

no matter how the data is formatted.

[<PATTERNS>]

0;0;0

0;1;0

1;0;0

1;1;1

Names of one or more of the patterns. Optional. Not required for import.

[<PATTERN NAMES>]

0 AND 0

0 AND 1

1 AND 0

1 AND 1

Comments of one or more of the patterns. Optional. Not required for import.

[<PATTERN COMMENTS>]

Both inputs 0

Left input 0 - right input 1

Left input 1- right input 0

Both inputs 1

End key word. Optional at end of file during import. If encountered during import reading the file stops here.

[<END>]

There may any comment following the [<END>] key word until the end of the file.

Raw Lesson CSV File

Raw Lesson CSV File

Besides [Sectioned Lesson CSV Files](#) MemBrain also supports import and Export of Lesson data from/into raw CSV files.

Raw Lesson CSV files contain less information than Sectioned Lesson CSV files but are much simpler in format and thus are easier to handle in conjunction with spread-sheet programs for example.

A Raw Lesson CSV file does not contain any section IDs or other information besides the actual data. It just contains values separated by the locale separator character according to your computer system settings (either ',' or ';' depending on the country setting of your computer).

The Raw Lesson CSV file always starts with the names of the input and output neurons according to the current lesson opened in the Lesson Editor. I.e. if the Lesson Editor is adjusted to capture the data of for example five input neurons and two output neurons then an overall number of seven comma separated names is read from the Raw Lesson CSV file upon import. A carriage return/line feed must follow the last neuron name.

After that the Pattern data of the lesson follows. As with the I/O neuron names the pattern data is read from the file in packages of a size that makes up the sum of input and output neurons in the lesson editor where every complete pattern must be followed by a carriage return/line feed.

Note that there may be additional carriage return/line feeds within the data of one pattern and also in the neuron names. Also all blank lines are ignored during import.

In the following a **valid example** for a Raw Lesson CSV file is given. Assumption is that the currently edited Lesson in the Lesson Editor has three input neurons and two output neurons. Also it is assumed that the value separator character of the computer system is a semicolon ';' and the decimal separator is a

comma ','. This is the typical setting for a German computer system. Computers configured for USA for example typically use comma ',' as the value separator and a point '.' as the decimal point separator.

MemBrain will automatically use the setting of your computer system during import and export of CSV data. This is something you have to consider if you want to use CSV files created on computers with a different country setting than your's. That applies to all CSV files created or imported by MemBrain not just the Raw CSV version.

Sample content of a RAW Lesson CSV file:

```
ln1;ln2;ln3;out1;out2
0,5;1;102,45;-0.6575;2,5
```

The same data would be also valid in the following format.

```
ln1;ln2
ln3;out1
out2
0,5;1;102,45
-0.6575;2,5
```

To import or export Lessons in the RAW Lesson CSV file format use the menu items

<File><Import Current Lesson (Raw csv)... resp.
<File><Export Current Lesson (Raw csv)...

in the [Lesson Editor's](#) menu.

Note: Since version 03.00.01.00 MemBrain also supports separate import and export of input and output data. In this case only the adjusted number of the accessed data is relevant, i.e. for importing the input data of a lesson only the number of output neurons adjusted in the Lesson Editor doesn't matter and vice versa.

To separately import/export input or output data use one of the following menu items

<File><Import Current Lesson Inputs (Raw csv)...
<File><Export Current Lesson Inputs (Raw csv)...
<File><Import Current Lesson Outputs (Raw csv)...
<File><Export Current Lesson Outputs (Raw csv)...

in the [Lesson Editor's](#) menu.

Normalize I/O Data

Internally MemBrain handles ranges of neuron activations between -1 and 1 or 0 and 1 depending on the activation functions of the neurons.

Real world data normally has data ranges different from that.

In MemBrain input and output neurons have the capability to automatically normalize I/O data to adjustable ranges. I.e. you can select the logical input range of an input neuron respectively the output range of an output neuron to best fit your original data.

Normalization can be adjusted both **manually** or supported by a **Wizard** for every input and output neuron separately. Manual adjustment is described together with the [neuron properties](#).

In MemBrain there is a **Normalization Wizard** available that simplifies the process of adjusting the normalization thresholds for your input and output neurons. To start the wizard, load a Lesson into the Lesson Editor that contains all data that will be used for training and validation of your neural net. Alternatively, if you don't have your training/validation data collected already you can create a new Lesson that has at least two Patterns where for every input and output neuron the minimum and the maximum applied activation value is entered.

Important Note:

It is essential that you provide the wizard with a lesson that incorporates the absolute maximum and minimum values expected for every input and output neuron. The resulting net will be limited to these data ranges and if you train the net then the training will also only be valid for these data ranges. This means that you also have to think about possible future data values when adjusting the normalization ranges of your net. Certainly, you can re-adjust the normalization ranges later on. However, please note that this will require you to re-train your net, since the training results are only valid on basis of a fixed normalization range for every input and output neuron.

After having loaded your lesson select all input and output neurons that shall be adjusted using the wizard. In most cases this will be all input and output neurons of your net, however, there may be situations where you might only want to change the settings for a subset of them. Note that it doesn't matter if hidden neurons are selected, too. Thus, if you want to edit the normalization settings of all input and output neurons you can just press <CTRL> + 'A' on the keyboard to select the whole net.

With the target neurons being selected and the Lesson being loaded execute **<Extras><Normalization Wizard...>** from MemBrain's main menu.

The following dialog will open.

You will see the name of the first edited neuron in the top of the dialog, its name is 'Temperature' in the given example. It is also stated that it is an input neuron.

On the right hand side you can see the absolute minimum and maximum values MemBrain has found for this input in the loaded lesson. According to these data MemBrain now suggests to use normalization for this neuron since the data range exceeds the normal range of the activation function of this neuron. Also, MemBrain already suggests a normalization range for this neuron on the left hand side (0 to 79 in this case).

If you are satisfied with this suggestion you can simply click <Next> to edit the next neuron. If you want MemBrain to use different values then just enter them into the corresponding fields and then click <Next>.

Once all neuron normalization settings have been determined the button <Apply> will get active. Click this button if you want all the normalization settings to be applied to the input and output neurons of your net. If you don't want to apply any changes to any neuron then select <Cancel>. MemBrain will not perform any edits to any neuron unless you finally click <Apply>.

Like any other operation on the net you can Undo all the changes performed with this wizard through the

[Undo function.](#)

Important Note:

The normalization wizard always suggests whole numbers for the range of a neuron. However, this might not always be the best solution. If your input data range is very small then you should consider choosing a tighter range around your minimum and maximum values using floating point numbers instead of whole numbers. The goal should always be to select a range that is large enough to incorporate all possible values but that is also as small as possible in order to make the best use of the activation function range of the neuron.

Handling Incomplete I/O Data

MemBrain is capable of handling incomplete patterns in your data (i.e. lessons) in a way that it can **ignore** certain **pattern output values** during the teaching process.

This is possible by specifying an '[Activation Ignore Value](#)' for **every output neuron** of the net that serves as a place holder for output data that shall be ignored during teaching.

Ignoring data during teaching may be useful if your data contains incomplete patterns, i.e. not all output data may be known for all of your patterns. In this case you can set the corresponding output data items of the patterns to this

special value so that they will be ignored during the teaching process.

Note that it is not possible to have the same feature for missing input data since input data must be set to some value during the propagation phase of the net ('Think Step'). Thus, if you have unknown input data in your patterns, best solution probably is to remove these patterns from your lesson. If you don't want to do this since it would affect too many of your patterns then the next best option probably is to set the unknown input values in your lesson to some average value with respect to the activation value range of the corresponding input neurons. However, choose the same value for this purpose in all of the patterns that contain unknown input values to these neurons. Another option certainly is to not use the corresponding input data, i.e. to remove the input neurons from the net and to remove the columns from the lesson.

Teaching a Net

When a net has been constructed and the [Net Analysis](#) does not detect any incompatibilities with the currently active teacher and if a [Lesson](#) is loaded in the [Lesson Editor](#) then the net can be trained to produce the [Patterns](#) defined by the currently loaded Lesson. This process is referred to as *teaching*.

In MemBrain teaching can be performed with different teachers. A Teacher represents a certain learning algorithm with certain parameterization. Teachers are administered and configured by the [Teacher Manager](#). Once the active teacher has been determined and configured in the Teacher Manager the [teaching procedure](#) can be performed.

When you use supervised learning then you should split your data into a training and a validation Lesson before to start to teach the net. Use the Lesson Editor to do so.

See [here](#) for more information on validating your net.

Selecting a Teacher/Training Algorithm

The currently active training algorithm (called 'Teacher' in MemBrain) can be selected via two ways:

- Use the drop-down box in MemBrain's 'Operation' tool bar:



- Use the [Teacher Manager](#)

Note that after the initial installation, only one single teacher is configured in MemBrain (The RPROP

teacher). In order to create and configure additional teachers use the [Teacher Manager](#). first.

Randomizing the Net

In order to achieve proper results during teaching a net has to be randomized before undergoing the first teaching process. Randomizing a net (or the [currently active sub net](#)) means that all the link weights and the neuron activation thresholds are initialized with small random values (unless the corresponding link or neuron properties are [locked](#)).

According to the above said, Randomization means that your net 'forgets everything'.

To randomize your net select <Net><Randomize Net> from MemBrain's main menu or click on the corresponding toolbar icon:

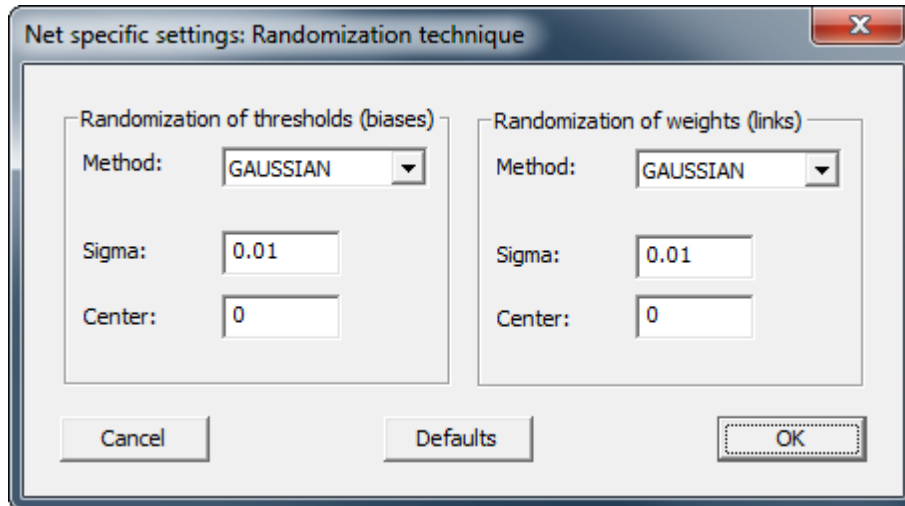


Randomizing will not influence the current activations of the neurons. Also, randomizing will not change the activation thresholds of input neurons as they are obsolete for such neurons because they have no input connector.

The randomization function of MemBrain can be configured individually for each developed net as described [here](#).

Configure Randomization

The randomization function of MemBrain can be configured individually for each developed net. I.e., the parametrization of the randomization function is stored together with the currently edited net. Choose <Net><Net Specific Settings><Randomization Settings...>. The following dialog will appear:



The randomization can be separately configured for neuron thresholds (left area of dialog) and link weights (to the right).

Available methods for randomization are:

- RANDOM
Standard random distribution
- GAUSS
Gaussian distribution

The Center of the distribution as well as its range/Sigma can be adjusted. The 'Defaults' button in the lower middle of the dialog brings back MemBrain's default values.

Shotgun Randomization

The 'Shotgun' randomization feature in MemBrain allows to randomize a net (or the [currently active sub net](#)) with an adjustable level of maximum impact. It can be used to help getting your net out of local minima: When your net error gets stalled and you feel that there might be a better global minimum of the error achievable then try to give your net a Shot with the Shotgun and start the training again. Starting from slightly different weights and threshold values the teacher might be able to pull your net out of a local minimum and into another, possibly better minimum.

To configure the Shotgun feature Select <Net><Configure Shotgun...> which will bring up the following dialog:

Configure Shotgun

Distribution

☐ Homogeneous

☒ Gaussian

Effect Method

☐ Percentaged

☒ Absolute

Homogeneous Parameters

Maximum Weight/Threshold Change:

Percentage: % Absolute:

Gaussian Parameters

Standard Deviation:

Percentage: % Absolute:

Center Value:

Percentage: % Absolute:

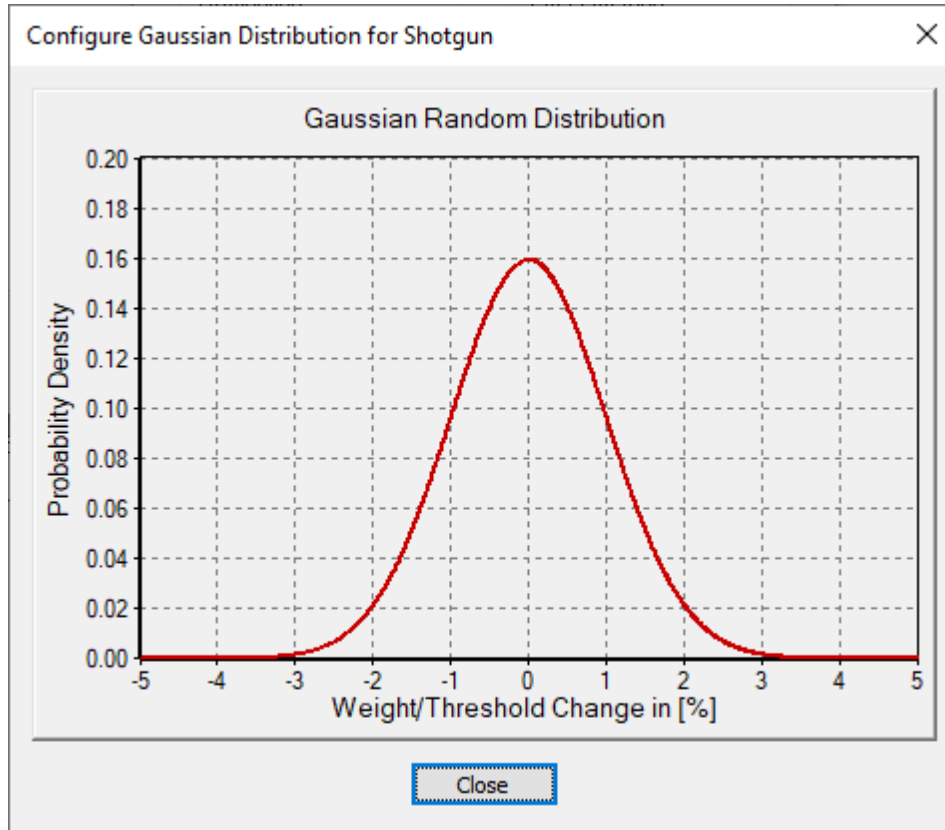
You can adjust the random distribution for the shotgun feature to either of the following


- **Homogeneous**
The random values are picked from a homogeneous distribution defined by a minimum and a maximum values. In between the minimum and the maximum the random values will be picked with identical probability.
- **Gaussian**
The random values are picked from a gaussian distribution defined by Standard Deviation and a Center Value.

Furthermore you can specify the Shotgun parameters to be either absolute weight/threshold changes or

relative (i.e. percentaged) ones.


If you select the gaussian random functionality you can plot the resulting probability distribution distribution with the corresponding button in the bottom of the dialog:



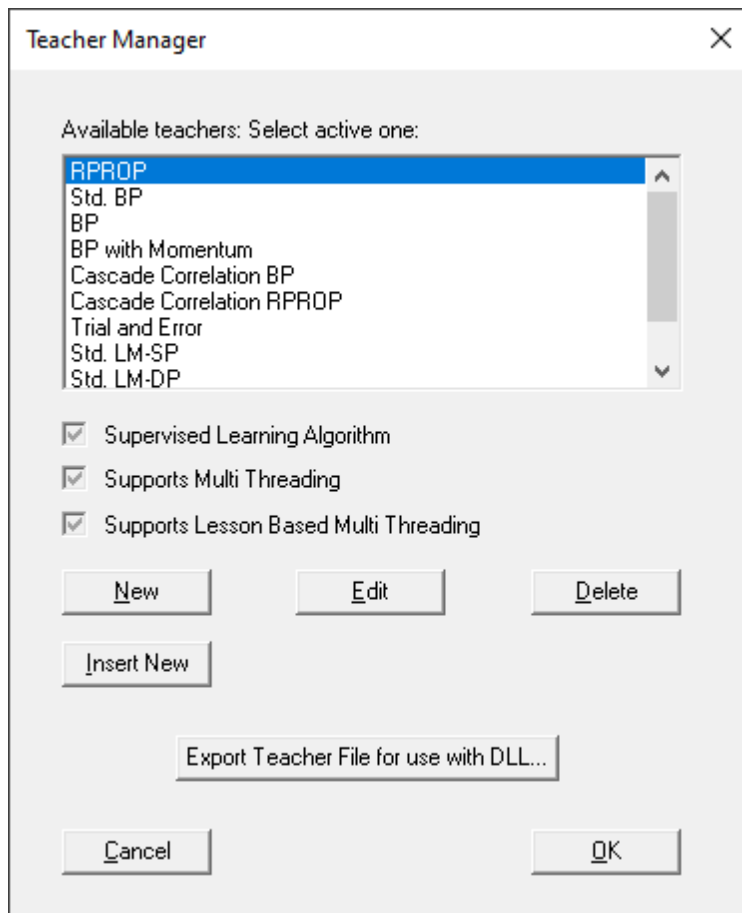
Whenever you use the shotgun on a loaded net (menu command <Net><Shotgun> or <Ctrl + G> or ) then all link weights and neuron activation threshold that are not locked will be changed by a random value according to the adjusted shotgun configuration.

The Shotgun feature can also be configured and used from within MemBrain scripts.

The Teacher Manager

MemBrain's Teacher Manager can be opened by selecting <Teach><Teacher Manager...> from the main menu or by using the tool bar button .

The Teacher Manager looks like this:



It gives an overview of the configured teachers and the selection bar determines the currently active one that will be used for teaching.

You can add a new teacher at the end of the list or insert a new one before the currently selected one by clicking on the buttons <New> respectively <Insert>. Teachers can also be edited by selecting them in the list and then clicking on <Edit>.

To delete a teacher select it in the list and then click on <Delete>.

With the button named 'Export Teacher File for use with DLL...' you can export MemBrain's teacher file to any location if you want to use the [MemBrain DLL](#) for training neural nets. The MemBrain DLL provides an interface function to load a teacher file and to select the active teacher from within your application that uses the DLL.

The Teacher Manager dialog also indicates whether the selected teacher

- Implements supervised training
- Supports [multi threading in general](#)
- Supports [lesson-based multi threading](#)

When a new teacher is created or an existing one is edited then the teacher editor dialog opens:

Edit Teacher

Name:

Type:

☒ Supervised Learning Algorithm

Learning Rate: Repetitions per Lesson: Repetitions per Pattern:

Target Net Error (for Auto Teacher):

☐ Online Learning (Batch Learning if not checked)

☒ Use Lesson ☒ Re-Apply Pattern (when repeating Patterns)

☒ Wait for Weblink Data Reception

☒ Use Weblink Sync

☐ Enable Weblink Remote Control

☐ Reset Net Before Every Lesson

☐ Rename Winner Neurons According to Patterns

☒ Use On-The-Fly Net Error Calculation if possible

Lesson Pattern Selection

☐ Ordered

☐ Random Selection

☒ Random Order

The following items can be adjusted/edited

Name:

You can give the teacher a name that will appear in the teacher manager's teacher list to reference the teacher.

Type:

The type of the teacher. For a list of currently available teacher types in MemBrain see [here](#).

Learning Rate:

The learning rate used in the algorithm of the teacher (if applicable). In case the teacher defines different start and end learning rates this is the start learning rate.

Repetitions per Lesson:

This parameter defines how often a whole [Lesson](#) with all its [Patterns](#) is taught when performing a single Teach Step in MemBrain. The Teach Step is the smallest unit that is executed by a teacher and thus also defines how quickly a teacher running in [Auto Mode](#) can be stopped without [interrupting the teach process](#) somewhere in between teaching the lesson.

Repetitions per Pattern:

Defines how often a Pattern is taught before the teacher switches to the next Pattern of the Lesson.

Besides potential effects on the teaching result this parameter, together with the Repetitions per Lesson parameter influences the time a teacher needs for a single Teach Step.

Target Net Error:

Whenever the teacher is running in [Auto Mode](#) the net error over the whole Lesson is compared to that value after every single Teach Step (defined by the repetitions parameters above). When the net error is \leq the Target Net Error the teacher will stop automatically and a message box will be displayed indicating that the Target Net Error has been reached.

Note: This only applies for supervised teachers. For non-supervised teachers this field is inactive.

Advanced:

This button is only enabled if the selected teacher type supports advanced settings. Another dialog will open that allows to configure these additional settings for the teacher.

Online Learning:

If this box is checked the weight and threshold changes determined by the teacher during the teach process are applied directly after training every single Pattern. If the box is unchecked this means that the teacher is operating in batch mode: The changes are accumulated over a whole Lesson and applied to the links and neurons afterwards.

Re-Apply Pattern:

This determines if a pattern is re-applied to the inputs of a net every time the pattern is repeated (only effective if Repetitions per Pattern > 1). If not checked the pattern is only applied to the inputs of the net once when the teacher switches to the next pattern.

Use Lesson:

When this option is checked then the teacher uses a lesson for operation. If unchecked the teacher operates without a lesson. In the latter case the data currently loaded in the Lesson Editor is of no effect. Not using a lesson with a specific teacher is for example applicable if the net operates with extern neurons (see [here](#) for more information) and if another MemBrain instance is controlling the lesson that applies to a net that is distributed over several computers.

Lesson Pattern Selection (3 options)

With these three radio buttons you can select how the patterns of a lesson are selected during teaching:

1. Ordered:
The order of patterns picked for teaching is the same as defined by the lesson. The teaching starts with the first pattern of the active lesson in the lesson editor and ends with the last pattern.
2. Random Selection:
Patterns are selected from the lesson randomly. There might be patterns taught multiple times while others may not be selected at all during one teach step (one 'Epoch').
3. Random Order:
Like Random Selection but every pattern of the lesson is picked exactly once during every teach step. Only the order of the patterns is determined randomly.

Independently from the chosen Lesson Pattern Selection option the view is always updated after teaching the currently active pattern in the [Lesson Editor](#). Thus, by changing the active pattern in the Lesson Editor using the Up/Down arrows you can watch the reaction of the net to different patterns during teaching on the screen. This does not influence the order of patterns during teaching.

Wait for Weblink Data Reception:

If activated then the teacher waits for teacher data requested over the Weblink to be received before processing the next teach step.

Use Weblink Sync:

With this option activated the teacher waits for every other connected MemBrain instance to complete its current teach step before it performs its own teach step. Through this option a synchronization on basis of single teach steps is achieved between several MemBrain instances connected over [TCP](#).

Enable Weblink Remote Control:

If the currently active teacher of MemBrain has this box checked then the teach process of this MemBrain instance can be remotely started and stopped by other MemBrain instances.

Reset Net Before Every Lesson:

If the currently active teacher of MemBrain has this box checked then the net will be automatically reset before every lesson run during teaching. Resetting the net means that all neuron activations and activation spikes along the links are set to 0. Resetting the net does not affect the weights of the links or the neuron activation thresholds. This option is without noticeable effect when training Feed-Forward networks with link lengths of the value 1 only and when the Activation Sustain Factors of all hidden and output neurons are 0. However, if at least one of these characteristics is different for your net then this option significantly influences the training since the teacher always starts with the same internal net state for every lesson run if this option is set.

Rename Winner Neurons According to Patterns:

If this check box is activated with a teacher then the teacher will rename the winner output neurons during teaching with the name of the corresponding patterns as defined in the currently active lesson. This is a very useful feature during unsupervised training: If all patterns in the lesson are assigned meaningful names then the teaching of a SOM can be observed in real time: Every time a new input pattern is applied to the neural net during training the winner neuron (i.e. the neuron with the highest normalized activation value) will be assigned the name of the corresponding pattern. Thus, if the option <View><Update View during Teach> is activated, too, then it is possible to see the SOM inflate during training. Also, it becomes visible which pattern locates to which neuron in the SOM after the training.

Use On-The-Fly Net Error Calculation if possible:

If the Lesson that is used for training and the Lesson that is used for calculating the Net Error are identical (see [Lesson Editor](#) for selecting the Net Error Lesson) then an accelerated approach can be used to calculate the net error:

Instead of applying all Patterns of the Lesson again after one Training Run on the Lesson and then calculate the Net Error the Error is calculated 'On-The-Fly' during the training Lesson run already. This speeds up the whole training process but certainly only is possible if the training Lesson and the Net Error Lesson are the same. Also, this approach results in an incorrect Net Error value since the Net Error is not calculated based on a frozen net but the Net may change during training after each pattern. If you want to suppress the approach of 'On-The-Fly' Net Error calculation then remove the check mark from this box. The Teacher will in this case always apply a separate Net Error Run after the specified number of Lesson runs have been executed.

Note that in nets with loop backs or other time dependent behaviour this setting also influences the internal state of the net (since additional Think Steps of the net are performed for calculating the Net Error) and may have significant impact on the results!

Teachers in MemBrain

MemBrain supports supervised and non-supervised learning algorithms.

Learning algorithms in MemBrain are implemented as so called 'Teachers' which are administrated through the [Teacher Manager](#).

A Teacher is defined by a learning algorithm and a set of parameters that determine the behaviour of the Teacher.

See below for details on the different Learning Algorithms supported by MemBrain.

- [Supervised Learning Algorithms](#)
- [Non-Supervised Learning Algorithms](#)

Supervised Learning Algorithms

Supervised Learning

MemBrain currently supports the following supervised learning algorithms:

- [Std. BP](#) (No Loopback support)
- [BP](#) (Full Loopback support)
- [Std. BP with Momentum](#) (No Loopback support)
- [BP with Momentum](#) (Full Loopback support)
- [RPROP](#) (Full Loopback support)
- [Std. Levenberg-Marquardt](#) (No Loopback support)
- [Cascade Correlation](#) (Full Loopback support, using BP with Momentum)
- [Cascade Correlation](#) (Full Loopback support, using RPROP)
- [Trial and Error](#) (Full Loopback support)

Standard Backpropagation

Std. BP (No Loopback support)

This teacher implements the standard backpropagation algorithm. [Loopback](#) links are not trained by this teacher. However, the net may contain loop back links. They are just left untouched by the teacher.

Backpropagation with Loopback support

Backpropagation (Full Loopback support)

For feed forward nets the effects are the same as with the Std. Backpropagation. However, if the net contains [Loop back](#) links then these are also trained. A special algorithm based on the previous output values of the neurons that feed back past values into the net is used to train the Loop back links.

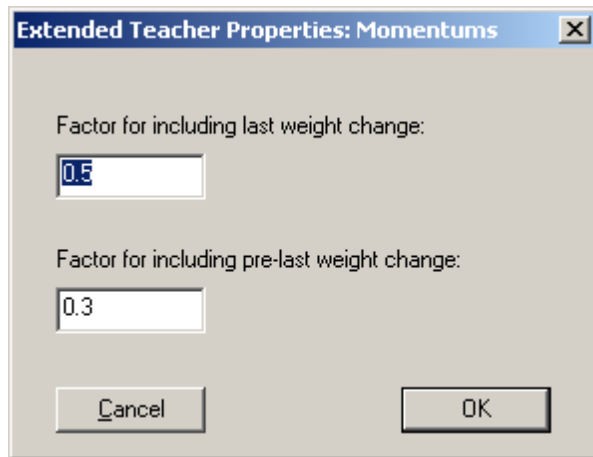
If you don't use loop back links in your net then you should use the Std. Backpropagation teacher instead since this is a bit faster.

Standard Backpropagation with Momentum

Std. BP with Momentum (No Loopback support)

This teacher Implements the standard backpropagation algorithm with an additional momentum term that brings in the past two weight or threshold changes, respectively. [Loopback](#) links are not trained by this teacher. However, the net may contain loop back links. They are just left untouched by the teacher.

The parameters for the momentum can be adjusted in the [Advanced](#) features accessible on the Teacher Managers's Teacher Editor. If you click on the <Advanced...> button on the [teacher editor](#) the following dialog opens.



You can specify the momentum terms of the teacher here. The momentum is defined by the factors the last weight change and the weight change before the last one are taken into account when calculating the next weight change of a link (or threshold change of a neuron).

Backpropagation with Loopback Support and Momentum

Backpropagation with Momentum (Full Loopback support)

For feed forward nets the effects are the same as with the Std. Backpropagation with Momentum. If the net contains [Loop back](#) links then these are also trained. A special algorithm based on the previous output values of the neurons that feed back past values into the net is used to train the Loop back links.

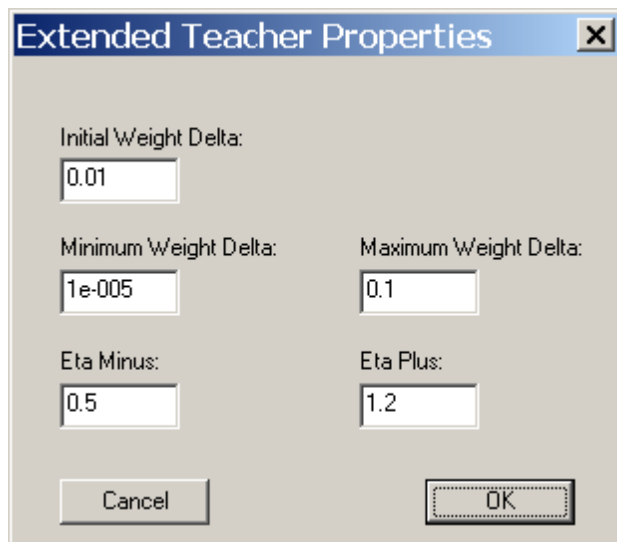
If you don't use loop back links in your net then you should use the Std. Backpropagation with Momentum teacher instead since this is a bit faster.

RPROP with Loopback Support

RPROP (Full Loopback support)

This teacher Implements the RPROP (Resilient Backpropagation) learning algorithm including support for loop back links.

The parameters for the teacher can be adjusted in the [Advanced](#) features accessible on the Teacher Managers's Teacher Editor. If you click on the <Advanced...> button on the [teacher editor](#) the following dialog opens.



You can specify the initial absolute value of the weight/threshold change as well as the minimum and the maximum weight change value that may be applied during teach.

The parameters 'Etha Minus' and 'Etha Plus' reflect the well known factors used by the RPROP algorithm to increase/decrease the absolute value of the applied weight change.

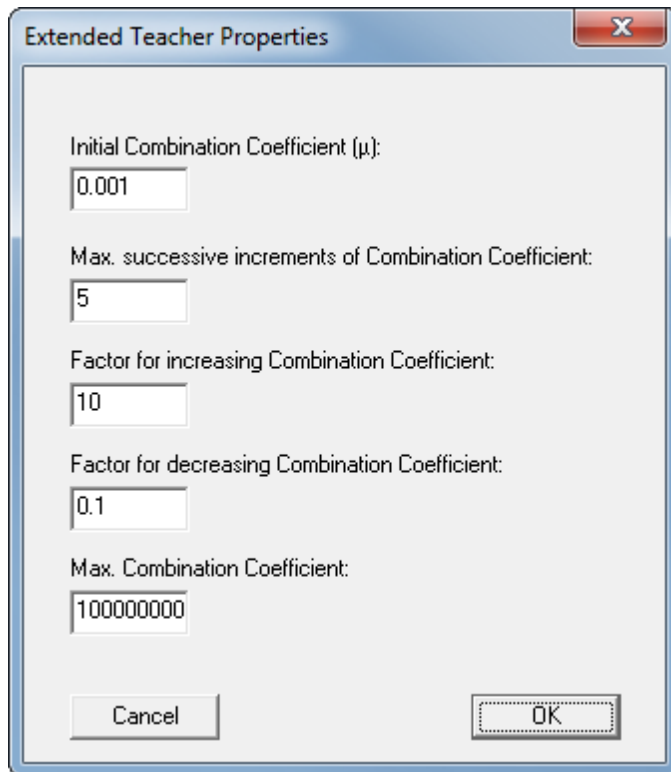
Note that the RPROP learning algorithm is only available in batch mode since the definition of RPROP is based on batch mode. Online mode is not defined for this kind of teacher.

Standard Levenberg-Marquardt

Std. Levenberg-Marquardt (No Loopback support)

This teacher Implements the Levenberg-Marquardt (LM) algorithm. [Loopback](#) links are not trained by this teacher. However, the net may contain loop back links. They are just left untouched by the teacher.

The parameters for the LM algorithm can be adjusted in the [Advanced](#) features accessible on the Teacher Managers's Teacher Editor. If you click on the <Advanced...> button on the [teacher editor](#) the following dialog opens.



Extended Teacher Properties

Initial Combination Coefficient (μ):

Max. successive increments of Combination Coefficient:

Factor for increasing Combination Coefficient:

Factor for decreasing Combination Coefficient:

Max. Combination Coefficient:

You can specify the combination coefficient (named μ in LM terminology) here as well as the rules for modifying μ as a result of the training process (increase/decrease factors). Additionally, the maximum number of successive μ increment steps and the limit value for μ can be specified.

This training algorithm is implemented in two versions in MemBrain: Single Precision and Double Precision. The Single Precision implementation uses 32 bit floating point arithmetics while the Double Precision implementation is based on 64 bit floating point numbers. Usually, one would select the Double Precision version as default and only switch to the Single Precision version in case MemBrain runs out of memory resources or in case training speed is crucial: Levenberg-Marquardt training in general requires significant memory and computing resources.

Cascade Correlation with Loopback Support

Cascade Correlation (Full Loopback Support)

The Cascade Correlation Algorithm actually is more than a simple training algorithm. Often it is also referred to as a 'Learning Architecture'.

This is because Cascade Correlation is an approach where not only links and thresholds of a neural net are trained but moreover the algorithm adds neurons and links during the teaching process as appropriate.

Normally, the Cascade Correlation Teacher is applied to neural nets that consist of only an input and an output layer, fully interconnected with links. However, the MemBrain Cascade Correlation Teacher is not restricted to this start architecture of a net, the teacher can start off with every already existing net and try to make the best of it.

The most significant advantage of Cascade Correlation is that you don't have to worry about how many hidden layers you should add to your net and how many neurons to place in each layer for an optimal result. Just start off with an input and an output layer, as said above, normally they should be fully interconnected through links. Then start the Cascade Correlation Teacher.

The teacher will first try to minimize the net error with the given architecture by adjusting links and thresholds in the same way as the [Backpropagation Algorithm with Momentum](#) or the [RPROP](#) does, depending on what version of the algorithm you choose. Once the net error does not change anymore for a certain number of Lesson Runs the teacher will add a set of so called Candidate Neurons to the net. In the first step the outputs of these units will not be connected to the net. However, the inputs of the candidates will be connected to every input neuron of the net and also to every hidden neuron already present in the net. The thresholds and the input links of the candidates will all be initialized by random values.

Now, the teacher will start a separate training run on the just inserted candidate neurons and their incoming links. The goal of this training is to maximize the correlation between the candidate neuron's activation and the residual output error of the net. In other words, the teacher trains the candidates to get active in situations where the net still shows significant errors on the outputs. Please note that this explanation is not absolutely correct, mathematically spoken. However, we want to focus on the basic functionality here.

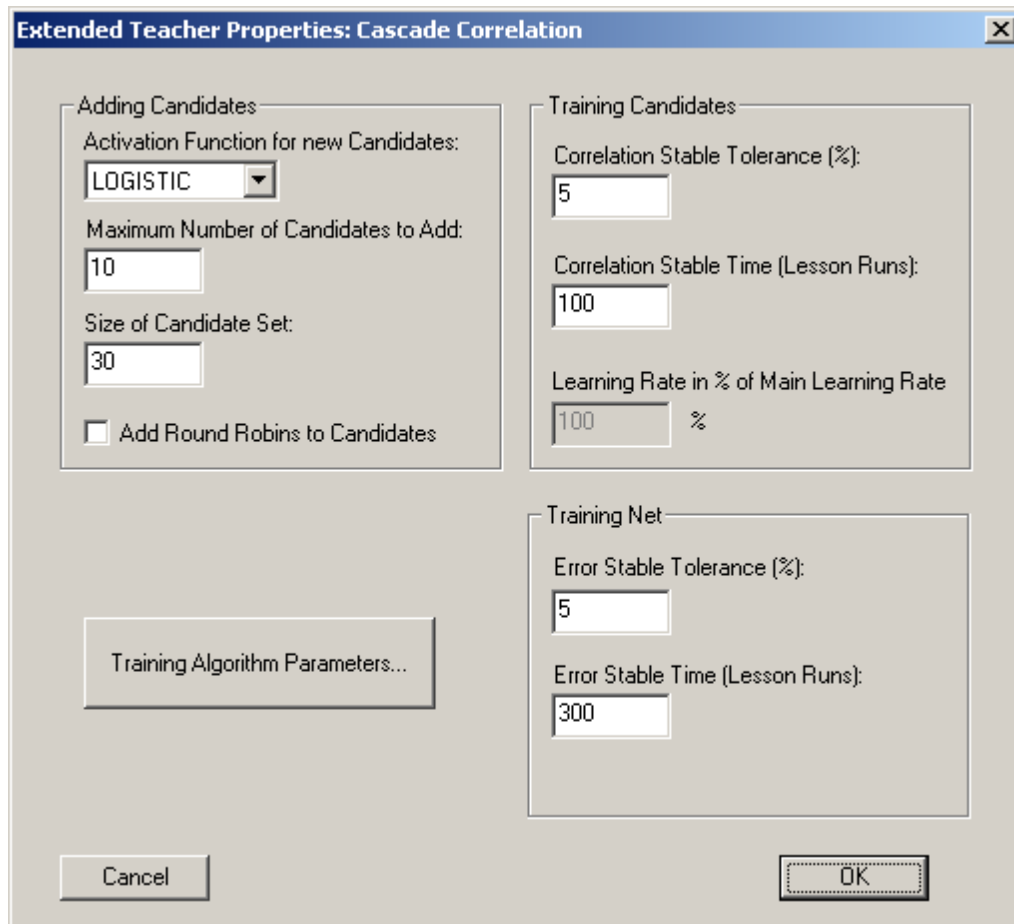
Using the above mentioned approach, the candidate units become so called 'Feature Detectors' i.e. they represent certain 'features' in the input data sets which couldn't be learned so far by the net.

During the candidate training process the teacher permanently supervises the course of the correlation value mentioned above for every candidate. Also the [Net Error Viewer](#) will change during this process to show the correlation value of the best candidate instead of the net error. If the correlation value of the best candidate stagnates then the teacher will delete all other candidates and install the best candidate permanently in the net. The input links of this neuron and also its activation threshold will be locked (i.e. frozen), and the output of the neuron will be connected to all output neurons in the net using randomized link weights.

A new training cycle on the net will start using the selected learning algorithm again. If the net error stagnates again the next set of candidates is added to the net and trained to become new feature detectors. This is continued until either the target net error is reached or until the maximum number of candidates have been installed in the net without reaching the target net error. Note that due to the fact that the feature detector's activation thresholds and their input links are locked these will not be affected by the normal training. The feature detectors are permanently installed in the net. This is another advantage of the Cascade Correlation Teacher: It is good for re-training a net on new patterns without the net forgetting the old patterns too quickly.

By the approach described above, every installed feature detector will become a new hidden layer in the net. I.e. after the training the hidden layers in the net all have one neuron only.

The advanced settings of the Cascade Correlation Teacher are adjusted through the following dialog:



The dialog box is titled "Extended Teacher Properties: Cascade Correlation". It is divided into three main sections:

- Adding Candidates:**
 - Activation Function for new Candidates: **LOGISTIC** (dropdown menu)
 - Maximum Number of Candidates to Add: **10** (text box)
 - Size of Candidate Set: **30** (text box)
 - ☐ Add Round Robins to Candidates
- Training Candidates:**
 - Correlation Stable Tolerance (%): **5** (text box)
 - Correlation Stable Time (Lesson Runs): **100** (text box)
 - Learning Rate in % of Main Learning Rate: **100** (text box) %
- Training Net:**
 - Error Stable Tolerance (%): **5** (text box)
 - Error Stable Time (Lesson Runs): **300** (text box)

At the bottom left is a button labeled "Training Algorithm Parameters...". At the bottom right are "Cancel" and "OK" buttons.

The upper left section of the dialog allows to configure the process of adding candidate units. You can specify the activation function for the new neurons, the maximum number of new neurons to add during the teacher run and also the size of the candidate set to select the best candidate from. Also, you can specify if a Round Robin link will be added to every candidate. Please note that this feature is not recommended in most cases, however it might be useful when training recurrent networks. So if you are not sure, best choice is to leave it deactivated.

In the upper right section of the dialog you can specify the criteria for the teacher to stop training the current set of candidates and implement the fittest candidate into the net for the next backprop based teaching sequence. The Correlation Stable Tolerance specifies the percentage value the Correlation of the fittest candidate unit may fluctuate with respect to the last stable value so that the correlation is still defined as to be stable. The Correlation Stable Time defines how many Lesson Runs the correlation of the fittest candidate has to be stable until the candidate is implemented and normal learning starts again. Also, you can specify the learning rate that shall be used for the candidate training as percentage of the main learning rate of the teacher. Note that this field may be deactivated depending on the version of teacher you have selected. For example, the RPROP version does not support a learning rate since the RPROP algorithm doesn't define such a parameter.

The lower right area of the settings contains the corresponding options for the normal teaching sequences of the teacher. The only difference is that not the correlation of the candidates is supervised but the remaining Net Error.

I.e. if the Net Error is stable according to the given settings the teacher will add a new set of candidates and start to train them.

In the bottom left area of the dialog you can adjust the parameters of the normal learning algorithm component of the teacher. The dialog that opens up when you click on the button 'Training Algorithm Parameters...' will vary with the version of Cascade Correlation you selected. Please note that the settings made here apply both to the normal teaching process and also to the candidate teaching process.

Trial and Error

Trial and Error (Full Loopback support)

This teacher implements a trial and error approach to optimize the neuron thresholds and the links in the net. [Loopback](#) links are also trained by this teacher.

The training algorithm picks a random set of neurons and links and changes their thresholds and weights, respectively, based on random numbers. If the overall net error of the training lesson improved by this change then the teacher repeats the changes including a momentum term. If the net error got worse then the teacher reverts the changes of the last step and makes a new random based set of changes.

Non-Supervised Learning Algorithms

Supervised Learning

MemBrain currently supports the following non-supervised learning algorithms:

- [Competitive with Momentum \(WTA\)](#)

This learning algorithm can be used to train SOMs (Self Organizing Maps) in MemBrain.

Competitive with Momentum (WTA)

Competitive with Momentum (WTA)

This Teacher implements the 'winner takes it all' algorithm which is a non-supervised learning algorithm for training of SOMs (Self Organizing Maps), also called Kohonen Maps. The Teacher incorporates an adjustable momentum term.

The additional parameters of this teacher can be adjusted using the <Advanced...> button on the [teacher editor](#). The following dialog opens.

Extended Teacher Properties: Course and momentums

Learn rate and adaptation radius course
End learn rate as percentage of initial learn rate:
 %
Start adaptation radius (Default neuron width is 45):

End adaptation radius:

Stamping of Mexican Hat (0..1)

Max. radius for graph:

Momentum
Factor for including last weight change:
Factor for including pre-last weight change:

Width of Mexican Hat
Start: End:

Note: The graph shows the course of the neighborhood function for both the start of teaching (first lesson) and the end of the teaching (last lesson repetition). Both learn rate and mexican hat width fall in an exponential manner from start to end.

You can specify the learning rate in the end of the teaching process as percentage of the start learning rate.

Also you can set the adaptation radius for the beginning and the end of the teaching process. Outside of the adaptation radius around the winner neuron for the current input data pattern no adaptations of link weights are performed.

Besides the learn rate this teacher uses a so called Mexican Hat function to determine the intensity and direction of weight changes. The Mexican Hat function is visualized twice, once for the start of the teaching process and once for the end. If you change parameters you should always click on the button <Update Graph> to cause the graph to be updated with the changed values. The X-Scale of the graph is adjustable via the input field <Max. radius for graph>.

The width of the Mexican Hat is the point where the function goes through zero into the negative area. You can specify this point for the start and the end of the teaching process, always thought as radius around the winner neuron of the currently applied pattern.

There is an additional parameter that is called <Stamping of Mexican Hat>. This parameter defines how far the Mexican Hat function reaches into the negative area.

The end of the teaching is defined by the number of lesson repetitions specified in the [teacher editor](#). This teacher does not support continuous teaching by definition. Thus if you select this teacher it is not possible to start the teacher in Auto mode. Also it is not possible to view the net error as this is not defined when using non-supervised learning algorithms.

Please note that for successful operation of this Teacher your output neurons have to be adjusted to the activation function MIN_EUCLID_DIST which represents a maximum of the activation to be reached when the Euclidean distance between the input pattern and the input links of a neuron is minimal.

Teaching Procedure

Important Note:

Before you first start teaching a net you should randomize the link weights and neuron activation thresholds in the net as described [here](#)!

In MemBrain there are two operation modes in which a teacher can be used:

Single Teach Step Mode:

When you select <Teach><Teach Step> from MemBrain's main menu or click



on the tool bar then the currently selected teacher in the [Teacher Manager](#) performs one Teach Step: It teaches the currently loaded Lesson in the [Lesson Editor](#) with the settings that have been designed for the teacher in the Teacher Manager's [Teacher Editor](#).

Before teaching can commence, the net is checked by the [net analysis](#) if not already happened and it is checked that the Lesson loaded in the Lesson Editor is [in sync](#) with the net. These steps are performed automatically by MemBrain in background. You will only be notified about this process if an error occurred during the checks.

During teaching all features that could interfere with the teaching process are locked in MemBrain. Once the Teach Step has been finished these features are accessible again and the [Error Graph](#) is automatically updated (even if not currently activated).

Auto Teaching Mode:

Select <Teach><Start Teacher (Auto)> from MemBrain's main menu or click



on the tool bar to launch the [currently active teacher](#) in Auto Teaching Mode. As with the Single Teach Step Mode some checks are performed in background before teaching is started and you will be informed about errors resulting out of these checks.

During teaching all features that could interfere with the teaching process are locked in MemBrain. Once the teach process has been stopped these features are accessible again.

When using [supervised teaching algorithms](#) the [Error Graph](#) is automatically updated after every Teach Step performed by the teacher (even if the Error Graph is not currently activated). Teaching stops automatically if the [Target Net Error](#) adjusted for the teacher has been reached.

The teacher can also be stopped manually after every complete Teach Step using the command <Teach><Stop Teacher (Auto)> from the main menu or by selecting

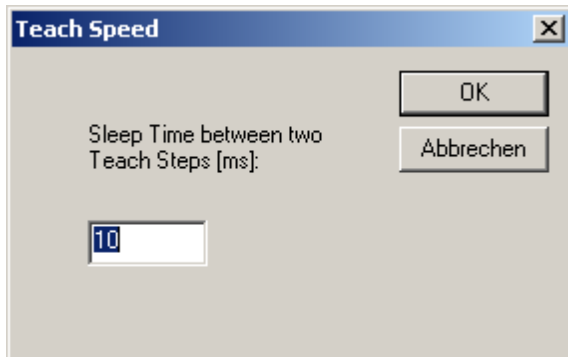


from the tool bar.

You can also abort the teaching process immediately by selecting <Teach><Abort Teacher (Auto)> from the main menu. When using this command keep in mind that the teacher will be interrupted immediately even in the middle of teaching a Lesson. Thus you should use this command only if stopping the teacher normally takes too long because of the size of the net and the Lesson.

Note: During teach you can always see the current net error displayed in the status bar of MemBrain's main window.

You can customize the speed of the teaching process by inserting sleep times in between every pattern during teach: Select <Teach><Set Teach Speed> from the main menu. The following dialog will open.



Enter the time in ms that MemBrain shall sleep between each two teach steps.

Note that this feature is only of interest for demonstration purposes. If the teaching process of MemBrain takes too much of your computing time so that other tasks do not get enough resources then it is a much better option to adjust [MemBrain's process priority](#) instead!

The Error Graph

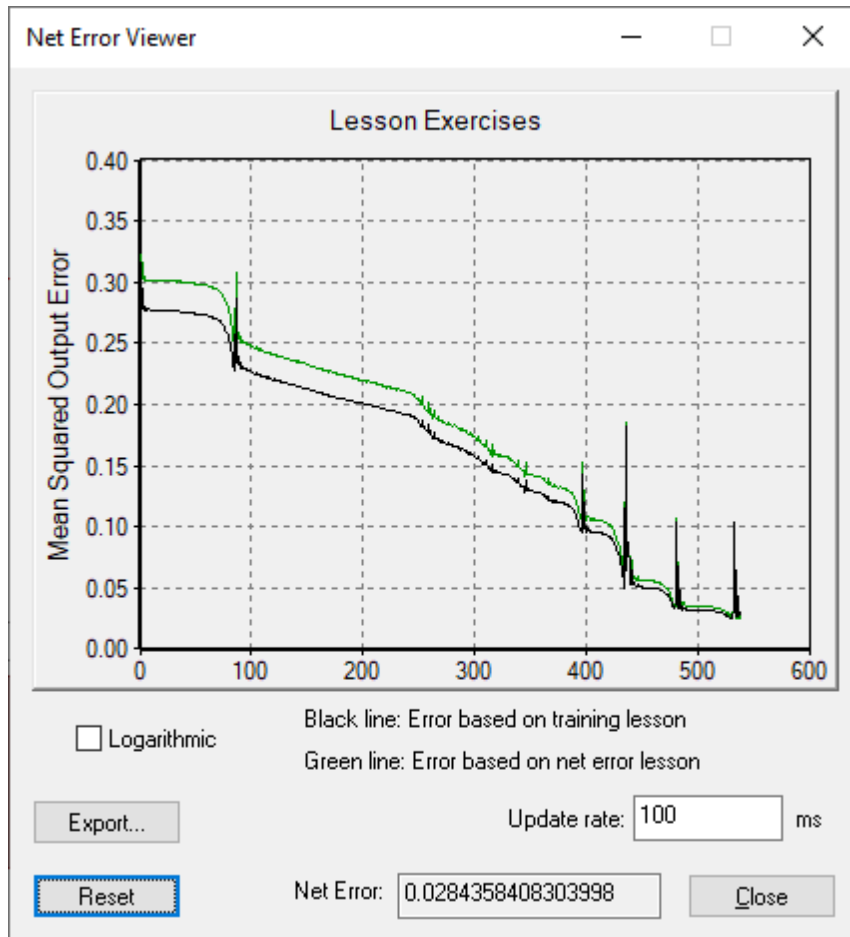
The Error Graph in MemBrain is a very useful feature to keep track of the current net output error and thus of the success of the teaching process.

If you have set a supervised teacher as the active teacher in the [Teacher Manager](#) the Error Graph can be invoked at any time by clicking



on the tool bar or by selecting <Teach><Net Error Viewer> from the main menu.

The Error Graph looks like following:



The graph shows two error curves in parallel:

- The net error based on the training lesson (black line)
- The net error based on the net error lesson (green line)

If the net error lesson and the training lesson are set identical (see [Lesson Editor](#) for more details on setting the training and the net error lesson) then only the black line is plotted.

The graph always auto adjusts to fit the whole error trace. You can reset the trace by the button <Reset> on the dialog at any time. This causes the trace to be deleted and started newly with a minimum axis range.

The trace is updated even if you close the Error Graph Viewer or even did not open it at all. Thus you can click away the window when not needed and re-open it again when you want to take a look at the past error history of the net during teaching.

The scaling of the Y scale can be toggled between logarithmic and linear using the check box 'Logarithmic' below the graph area. The setting is persistent, i.e. saved and loaded by MemBrain when exiting and re-starting MemBrain.

The update rate of the graph is adjusted to 100 ms when MemBrain is started. You can adjust the update rate via the provided edit field. Higher update rates (i.e. lower ms values) reduce overall performance while the viewer is displayed.

Export to CSV:

By use of the button <Export...> you can export the content of the Error Viewer graph to a CSV file for further processing in a spread sheet or other data analysis program for instance.

Note that the CSV file is generated using the application wide [CSV file separator settings](#).

For information on how the net error is calculated in MemBrain, see [here](#).

For detailed analysis which patterns of your data show which deviation from the target value please refer to the [Pattern Error Viewer](#).

The Pattern Error Viewer

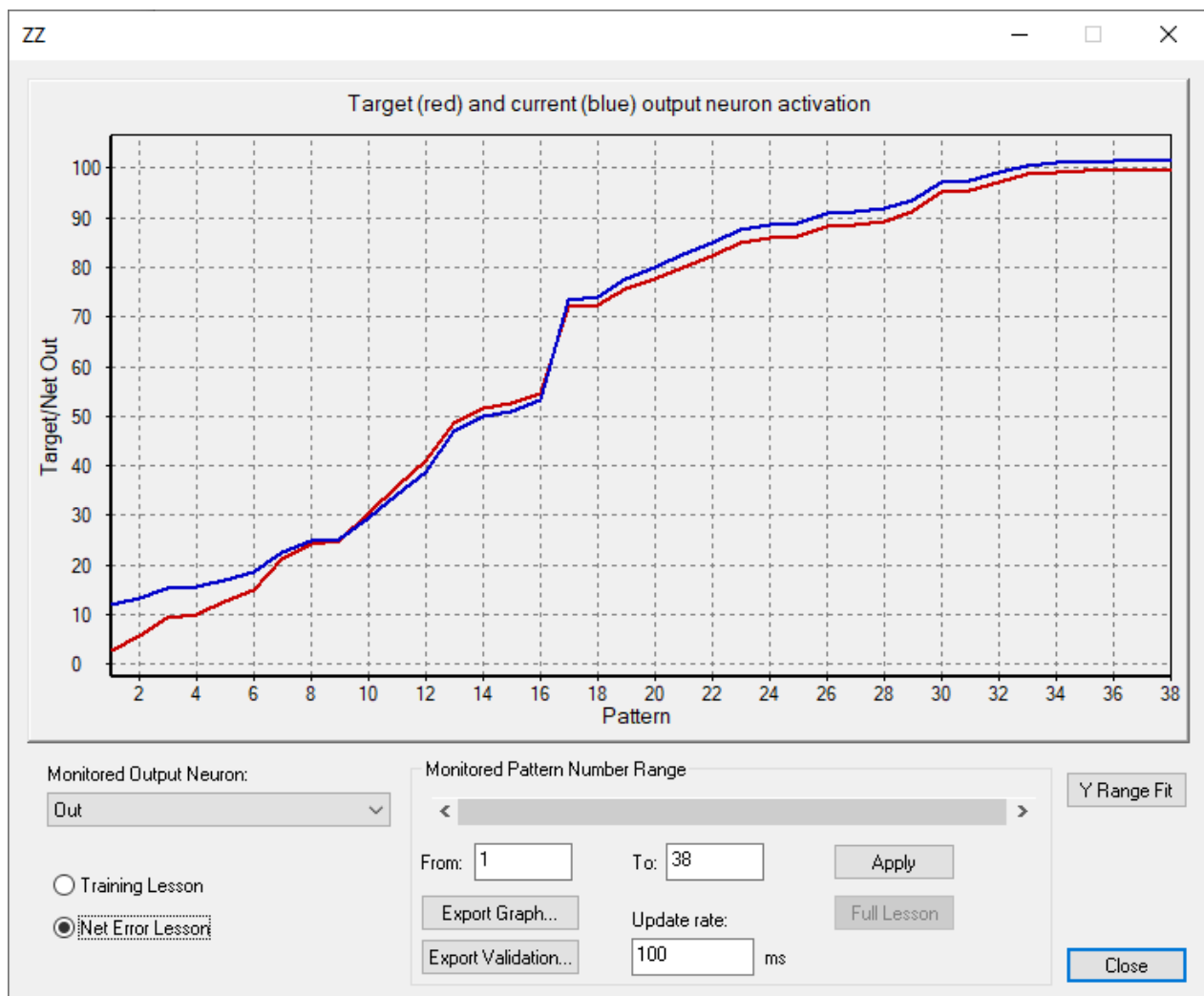
MemBrain allows you to monitor the activation of a single output neuron of your net as it performs over all data patterns of a whole lesson.

If you have set a supervised teacher as the active teacher in the [Teacher Manager](#) the Pattern Error Viewer can be invoked at any time by clicking



on the tool bar or by selecting <Teach><Pattern Error Viewer> from the main menu.

The Pattern Error Viewer looks like following:



The red curve in the graph represents the output activation as defined by the currently selected **Training Lesson** or **Net Error Lesson** as indicated in the [Lesson Editor](#). The blue curve shows the actual activation of the corresponding output neuron in the net.

The decision if the Training Lesson or the Net Error Lesson is displayed can be made by the radio buttons in the lower left area of the window. If Training and Net Error Lesson are set identical in the Lesson Editor then the option is displayed grayed and the radio button is automatically set to the option 'Training Lesson'.

This graph updates automatically during training so you can watch in real time during training how well your net output actually approaches the target as defined in the training or net error lesson, respectively. Also, the graph updates automatically if you select the command 'Think On Lesson' from the [Lesson Editor](#).

In the bottom left area you can select the output neuron to be monitored. The visible pattern range can be adjusted in the area 'Monitored Pattern Number Range'. If you select a pattern range smaller than the number of patterns in the lesson then the horizontal slider bar gets active and you can scroll through the whole lesson by dragging the slider with the mouse.

Mouse Wheel support in the Pattern Error Viewer:

- Scrolling with the mouse wheel will toggle through the drop-down box list of output neurons
- When pressing and holding the <Ctrl> key on the keypad scrolling with the mouse wheel will zoom in and out into the lesson data, i.e. modify the pattern range visible in the Pattern Error Viewer window.
- By tilting the mouse wheel to the left or to the right (not supported by all mice types) the scroll bar of the Pattern Error Viewer can be moved towards left or right one step.

The update rate of the graph is adjusted to 100 ms when MemBrain is started. You can adjust the update rate via the provided edit field. Higher update rates (i.e. lower ms values) reduce overall performance while the viewer is displayed

Export to CSV:

By use of the button <Export Graph...> you can export the content of the Error Viewer graph to a CSV file for further processing in a spread sheet or other data analysis program for instance.

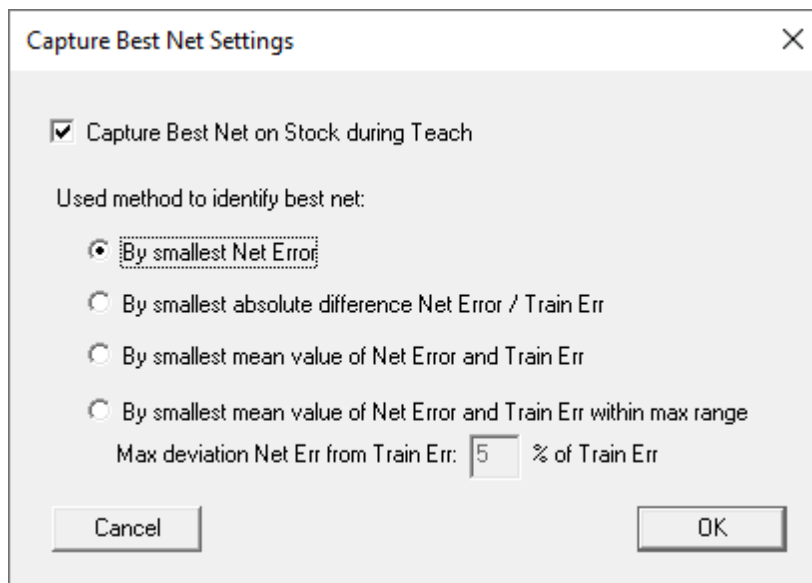
If you want to export all data, including the inputs of the lesson and also including all target and actual output data for the currently displayed lesson then simply click on the button 'Export Validation...'.

Note that the CSV files are generated using the application wide [CSV file separator settings](#)..

Auto Capturing Best Net on Stock

MemBrain supports automatic capturing of the best net version during training on the [Net Stock](#):

In the <Teach> menu select the option <Capture Best Net on Stock>.... In the dialog that opens, enable the option via the check box and select the method that shall be used to identify the best net during training:



MemBrain will then during the teach procedure always capture a copy of the current net on the stock if the selected criterion reaches a new minimum.

Note that when this option is active you cannot change the net error lesson in the Lesson Editor during the

teach process. This is to prevent the net/training error controlled best net selection to be rendered invalid.

After the teach process has finished MemBrain will automatically ask the user whether to re-load the captured best net from stock. However, this is only the case if no MemBrain script is currently running. When teaching is performed through scripting the net stock reload operation can be performed through the corresponding [script command\(s\)](#).

See [here](#) for more information on the Net Stock and how to re-load a net manually from the stock.

Using Group Relations to work with Sub Nets

It is possible to define relations of different types between groups of neurons in a net. This allows to define sub nets within a net which can then be trained separately using different algorithms and data sets.

This help chapter is structured into the following sections:

- [What are Group Relations](#)
- [Adding and Editing Group Relations](#)
- [Available Types of Group Relations](#)
- [Choosing the currently active Relation/Sub Net](#)

What are Group Relations?

Group Relations are used to define a relationship between a source group and a target group of neurons. MemBrain's [Grouping](#) feature is used as basis to define these relations.

Based on relations MemBrain allows to define sub nets within the net. I.e. a sub net is always defined via the following three elements:

- Source group
- Target group
- Relation that ties target group to source group

The relation itself is always defined via and owned by the source group. It provides certain properties which determine the kind of the relation and a set of parameters.

Once a relation has been defined, MemBrain allows to choose the currently active relation (i.e. the resulting sub net) so that the training process is applied to this area of the sub net only using the parametrization of the relation as guidance.

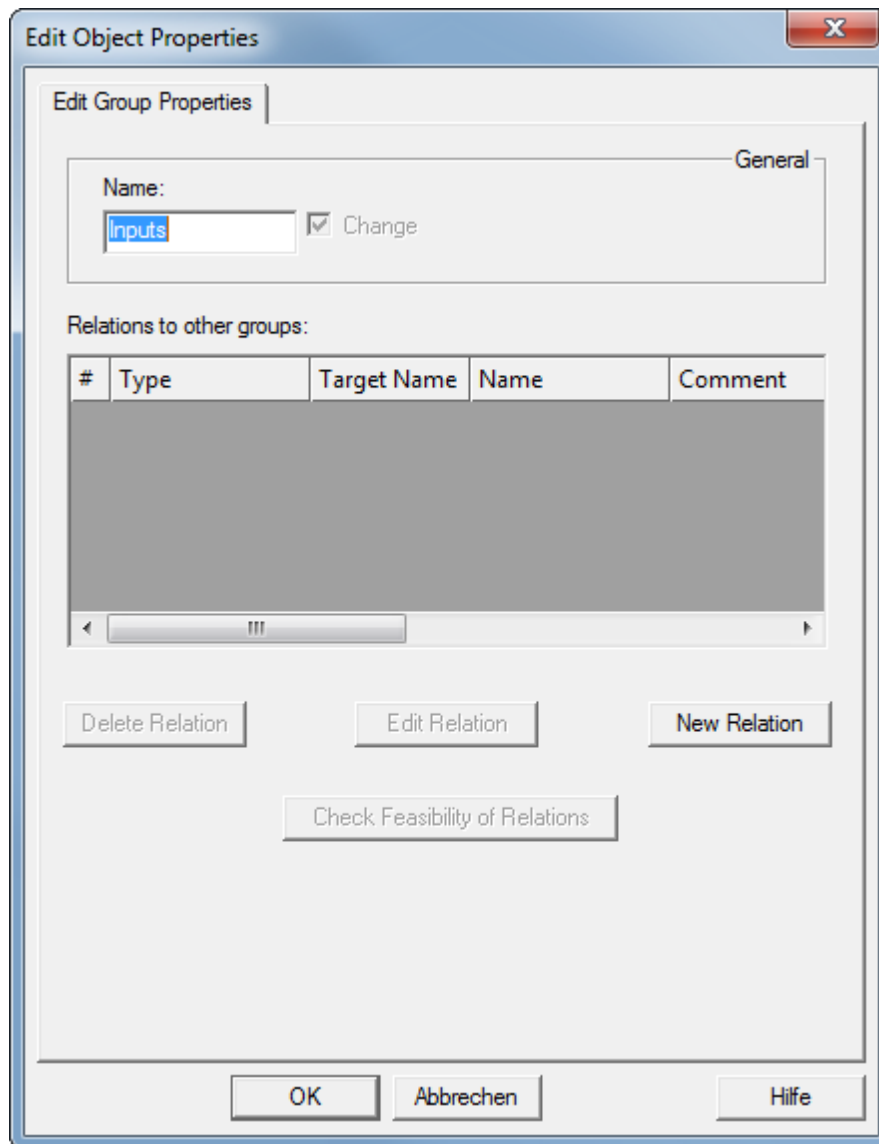
Through this, MemBrain for instance allows to define autoencoder sub nets which can be trained to reproduce input patterns in a hidden layer of the net. This is a way to achieve better generalization capabilities in a neural net.

See [here](#) on how to define and edit relations. [Here](#) you will find a list of available relation types and their usage.

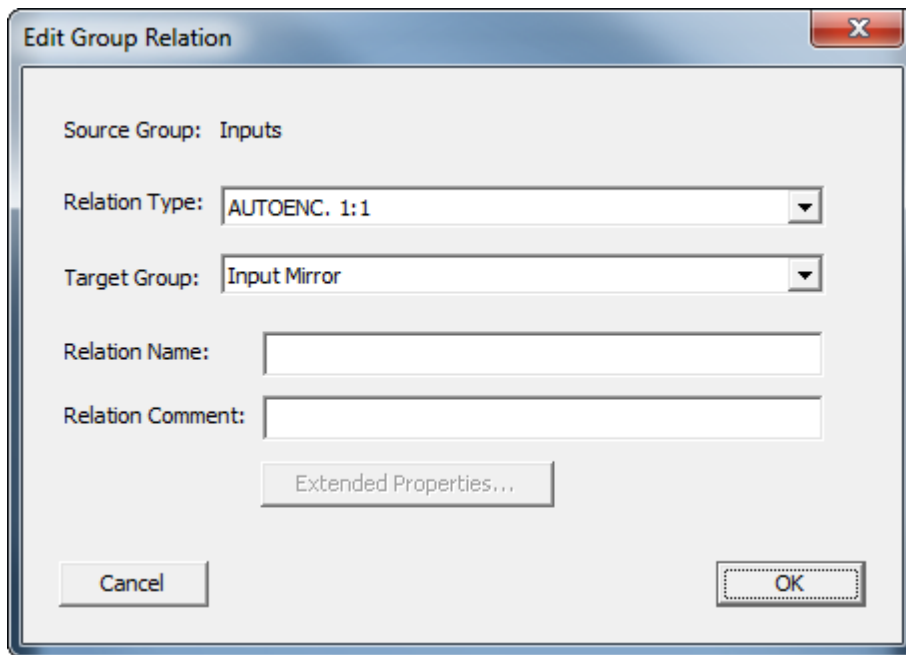
Adding and Editing Group Relations

In order to add a relation from a source group to a target group, do the following:

- [Open the group properties dialog](#) of the source group. In this example the source group has been named 'Inputs' since it contains all input neurons of the net. This is a user defined name however, it could be anything.
- In case you have not yet added any group relations the following dialog will come up:

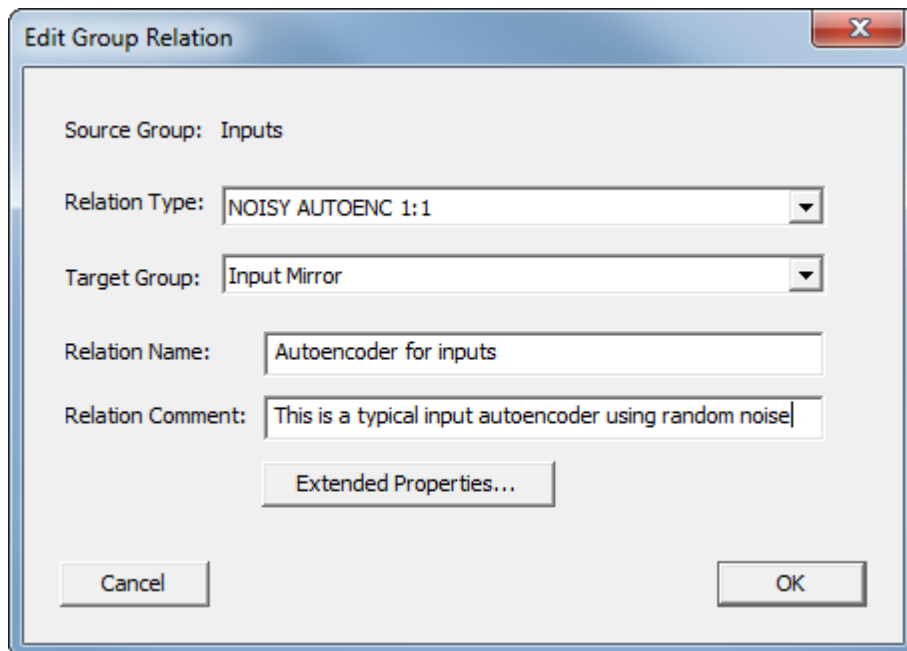


- Click the button 'New Relation'. Note: The button is only enabled in case there is at least one other group in the net.
This brings up the following dialog:



- As you can see the name of the source group 'Inputs' is provided on top of the dialog. Below you can select the type for the new relation from a drop down list. See [here](#) for more information about the available relation types and their implications.
- Below you will find a drop down list of all available other groups in the net. The example shows the preselected group named 'Input Mirror'. In the example this group contains the same number of neurons as our input neuron group, with the same layout (actually, it is a rectangular area of neurons forming pixels in a grey scale image).
- You can provide an optional user defined name and a comment for the new relation.

In this example we choose a different relation type to cause the button 'Extended Properties' to become enabled and we also provide a name and a comment:



- A click on the button 'Extended Properties' brings us to the following dialog:

The dialog box titled "Randomization technique" has a close button (X) in the top right corner. It contains two main sections:

- Distribution of random values:**
 - Method: GAUSSIAN (dropdown menu)
 - Sigma: 0.1 (text input)
 - Center: 0 (text input)
- Probability of randomizing a pattern:** 1 (text input)
- Probability of randomizing a specific input:** 1 (text input)

At the bottom, there are "Cancel" and "OK" buttons.

- Here we can adjust the properties of the noise generation for the new relation. We choose the following settings for this example

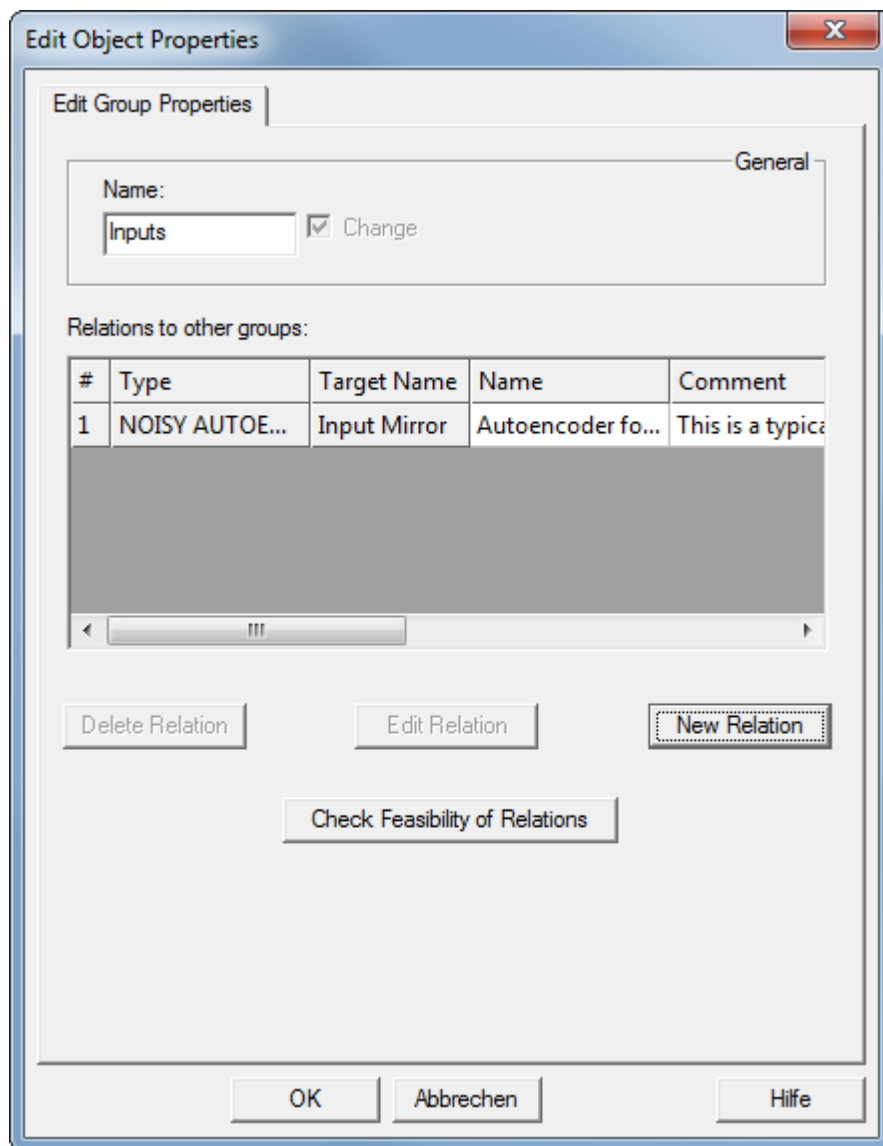
The dialog box titled "Randomization technique" has a close button (X) in the top right corner. It contains two main sections:

- Distribution of random values:**
 - Method: GAUSSIAN (dropdown menu)
 - Sigma: 100 (text input)
 - Center: 0 (text input)
- Probability of randomizing a pattern:** 0.3 (text input)
- Probability of randomizing a specific input:** 0.2 (text input)

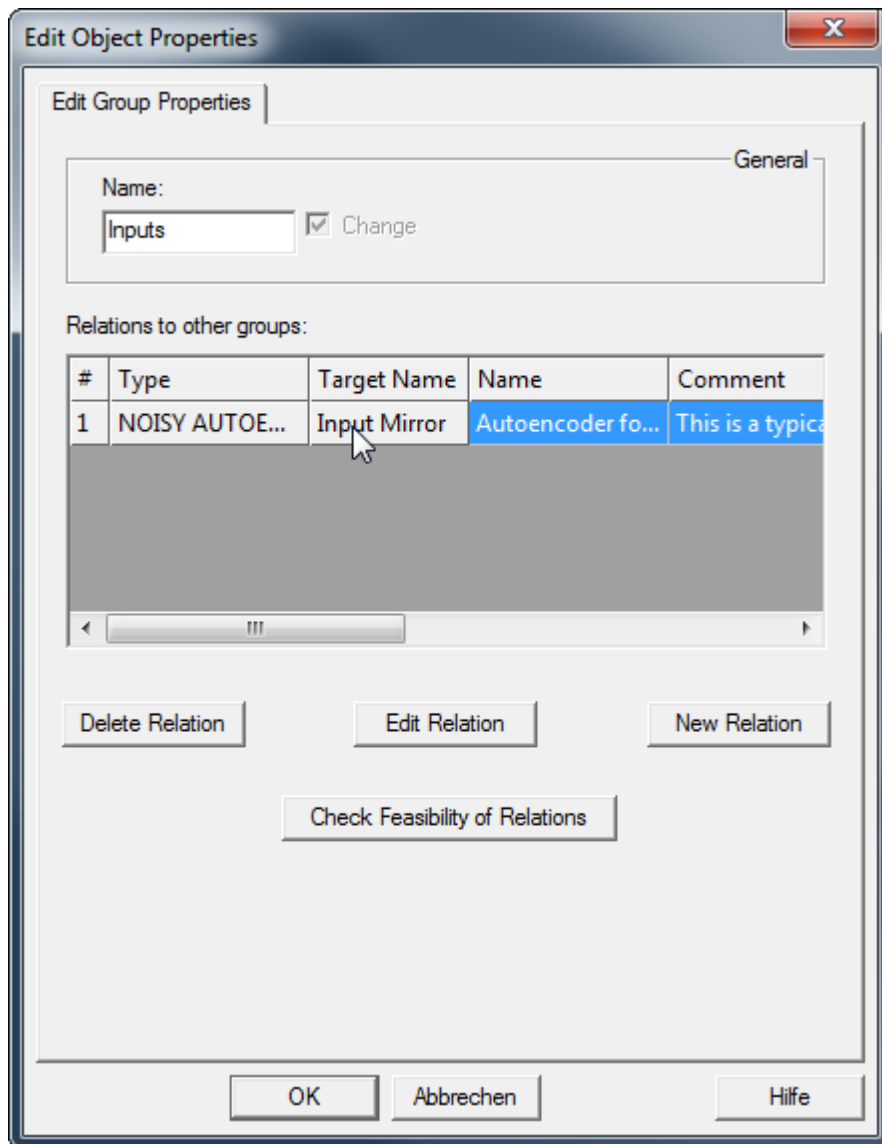
At the bottom, there are "Cancel" and "OK" buttons.

- Gaussian distribution of the noise added to each input pattern during training when using this relation
- The gaussian standard deviation 'Sigma' is 100. We choose this value for the example since our input neurons are adjusted to a normalization range of [0 .. 255]. So an added value of 100 represents a significant change in the brightness of the 'pixels'.
- The Center value for the gaussian distribution is set to 0
- The probability that a given pattern of the training lesson during training of the this relation is affected by random noise all (using the given Gaussian distribution') can be adjusted. We selected a value of 0.3 here. This means that only about every third pattern during training will be affected by noise.
- If a given patter is affected by noise then there is another probability that defines if the activation of a specific input neuron of the source group is affected by noise during applying this pattern. In our example this means that not all pixels (i.e. input neurons) are affected by noise during training even if noise applyance happens to a specific pattern.

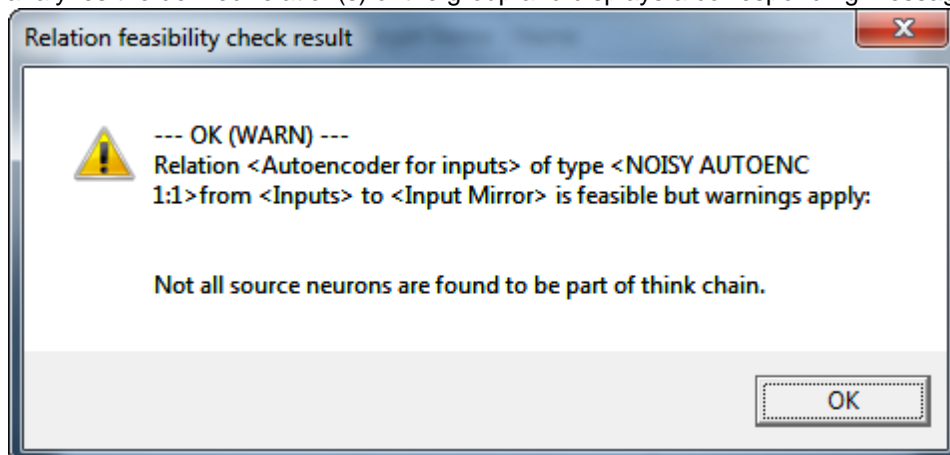
Clicking 'OK' two times brings us back to the group properties dialog where the newly added relation is displayed:



Note that selecting a row in the table of relations allows to edit or delete specific relations:

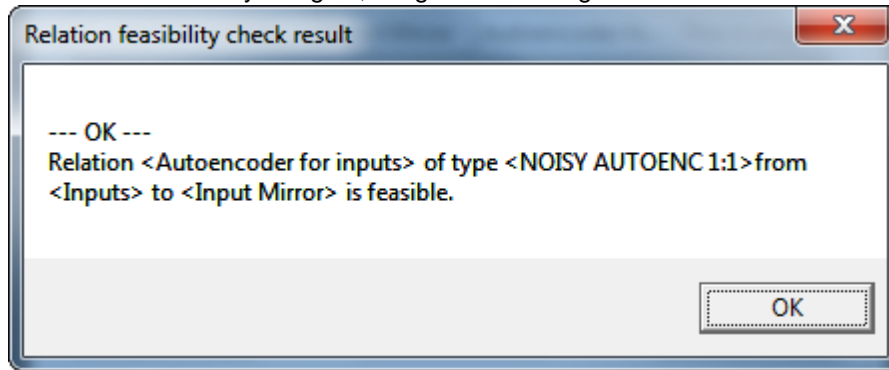


Note the button 'Check Feasibility of Relations' in the bottom of the dialog. When clicked, MemBrain analyzes the defined relation(s) of the group and displays a corresponding message. For example:



In this example MemBrain has found that the relation is feasible in general but a warning applies: The message says that not all neurons of the source group provide input (directly or indirectly) to the target group. Reason is that in the example above the neurons in our groups 'Inputs' and 'Input Mirror' are not interconnected at all.

If we close the dialogs, connect the neurons by links, either directly or indirectly via other hidden neurons and then run the analysis again, we get the message:



Editing an existing relation is easiest done by choosing it to be the currently active relation via the tool bar and then clicking the tool bar button named 'Rel...' besides the relation/sub net selection drop down box or by selecting the menu command <Edit><Edit Current Group Relation...>.



Available Types of Group Relations

MemBrain supports the following types of group relations:

- **AUTOENC 1:1**

Standard autoencoder: A sub net defined via this relation requires the same number of neurons in its source and in its target group. When trained, the sub net is optimized so that it learns to reproduce the data patterns applied to its source group by its target group neurons.

If the sub net contains one or more layers of hidden neurons between its source and its target group and if these layers contain significantly less neurons than the source and target group, then the net is forced to an advanced generalization of the data: The challenge for the net is to reproduce the source neurons activations by the target neurons but using a compressed data representation (the one from the last hidden layer before the target neurons).

A big advantage of this kind of training is that the data does not have to be labelled. Assume you have a large number of images which show handwritten numbers, like it is the case with the popular MNIST data set. Every single image represents one digit from 0 to 9.

for the autoencoder-based pre-training of the sub net it is not important which digit is represented by an image. MemBrain does not even require you to define this for autoencoder-based training. Instead, it is sufficient to have a lesson with only the inputs (i.e. the images).

- **NOISY AUTOENC 1:1**

Same as AUTOENC 1:1 but additionally some random noise is added to the activations of the source neurons during training. This forces the sub net to learn to reproduce the original source neuron activations (i.e. without noise) on its target neurons although some noise has been added to the source neurons during training.

Using a noisy autoencoder makes the net more robust with respect to noisy data. It is a way to generate additional training data vectors on-the-fly during training and thus helps to achieve better generalization results.

- **LESSON OUTPUTS TRAINING**

This relation type is used to train the target group of a sub net to the output data of a lesson.

It is often used in combination with an autoencoder relation: Assume that you have an autoencoder sub net in your net and this autoencoder sub net features a hidden layer which adjusts to a compressed representation of the input neurons during autoencoder training. You then typically would use a LESSON OUTPUTS TRAINING relation which is defined from the mentioned hidden layer (as source group) to the output layer of the net (as target group).

Training of the autoencoder is rather time consuming since typically many links are involved here. The

training of the lesson outputs is fast then, since it only involves the links between the hidden layer and the outputs which are typically far less.

- **NOISY LESSON OUTPUTS TRAINING**

Same as LESSON OUTPUTS TRAINING but additionally some random noise is added to the activations of the source neurons during training. This forces the sub net to learn to reproduce the lesson outputs on its target neurons although some noise has been added to the source neurons during training. Using a noisy autoencoder makes the net more robust with respect to noisy data. It is a way to generate additional training data vectors on-the-fly during training and thus helps to achieve better generalization results.

Choosing the currently active Relation/Sub Net

The currently active Relation / Sub Net can be selected via a drop down list box in MemBrain's Operation toolbar.

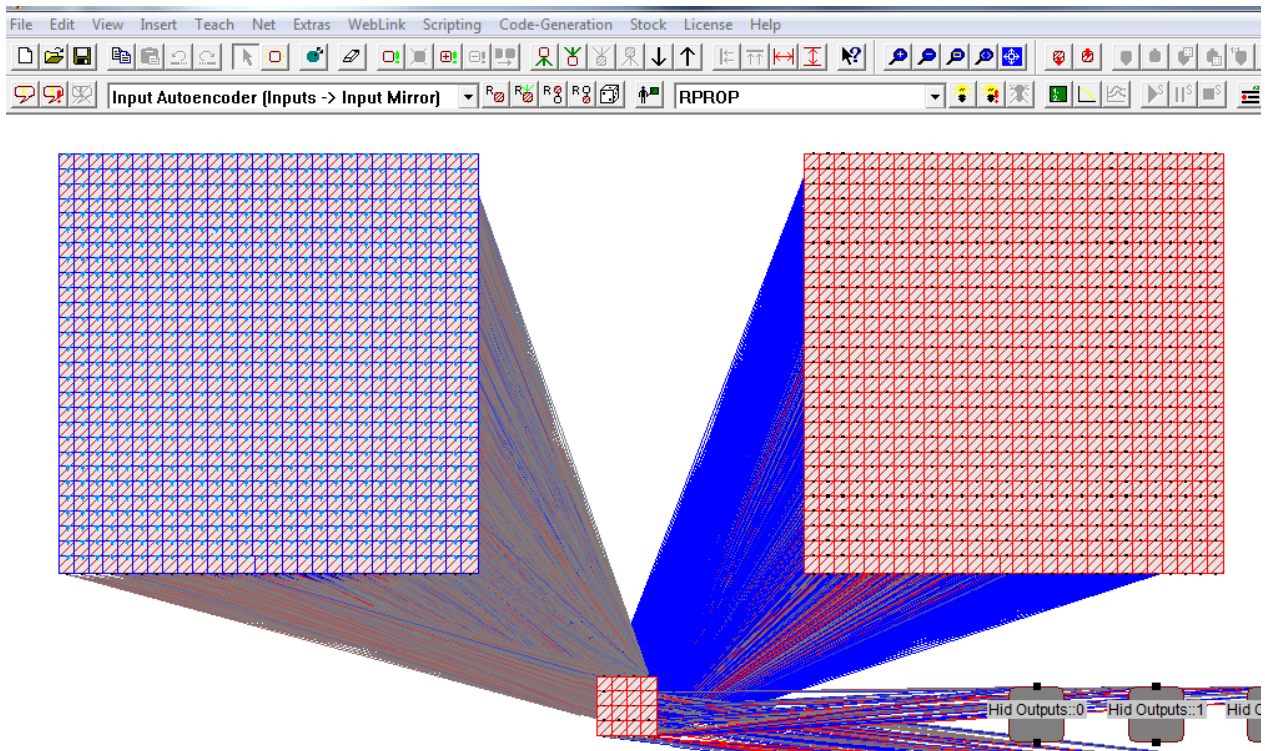
By default, and as long as there are no group relations defined in the net, the only available entry in the list is 'Full Net'. This means that training, when started, is applied to the full net as opposed to training of sub nets only:



Once group relations are defined the list allows to select these relations. This adjusts MemBrain's training to the sub net which is defined by this relation. In the example below you see a relation named 'Input Autoencoder' which relates the source group 'Inputs' to the target group 'Input Mirror'.



Once a group relation is selected from the list MemBrain automatically selects the neurons which are part of the sub net defined by this relation. In the example above this may look like shown in the following screen shot:







On the left hand side you see the neurons of the source group named 'Inputs' (which are input neurons in this case). On the right hand side you see the target group named 'Input Mirror'. This is an array of hidden

neurons of the same number and layout as the source group. The relation is defined from the source group to the target group. Note that the group names are not visible in the screen shot since the groups are uncollapsed.

MemBrain analyzes the relation and detects that the path of calculating the activations from the source group to the target group includes the 16 neurons shown in the bottom of the screen shot. I.e., MemBrain knows that these neurons belong to the defined sub net, too and automatically incorporates them in the required activities.

If at any time you need to select the neurons of the relation / sub net again you can use the following tool bar buttons:

-  selects the neurons of the sub net (same as selection in the list dows)
-  selects the neurons and the links of the sub net
-  selects the head/source neurons of the sub net
-  selects the tail/target neurons of the sub net

Convolutional Neural Networks

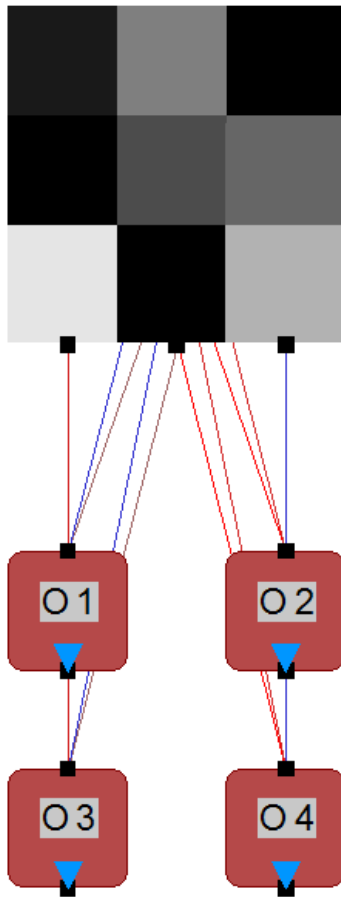
MemBrain supports so called "Convolutional" neural networks. These are neural nets with links and/or neurons that share a common weight or threshold, respectively.

Convolutional neural nets are often used in image recognition projects: A layer with input pixels is connected to the next layer using convolutional links, i.e. links that partially share common weights according to specific connection schemes.

A simple example for demonstration purposes is provided below:

- Input pixel matrix: 3 x 3 pixels
- Output layer: 4 neurons
- Connection scheme: Each 2 x 2 pixel area is fully connected to one of the output neurons.

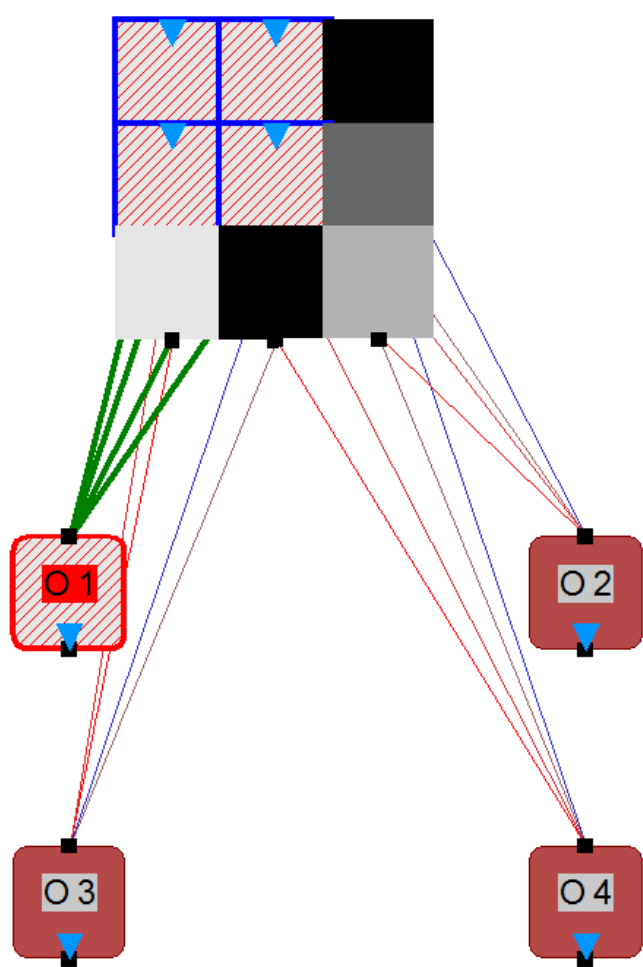
The full net looks like this:

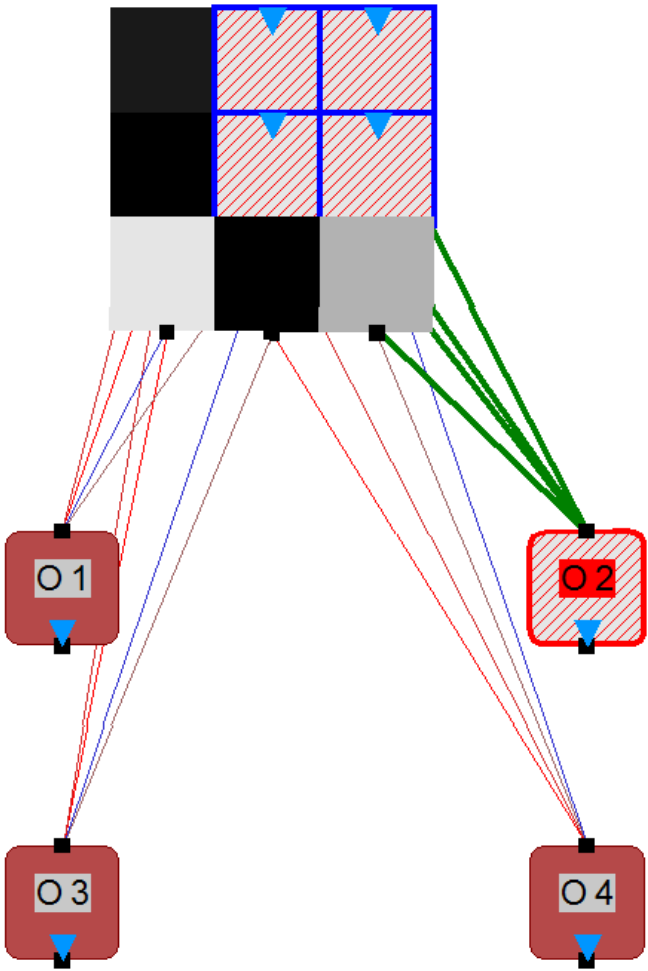


Pulling the output neurons a bit apart from each other and selecting the linkage (see [here](#) on more information on object selection functions), we can see the connection scheme:

Part1

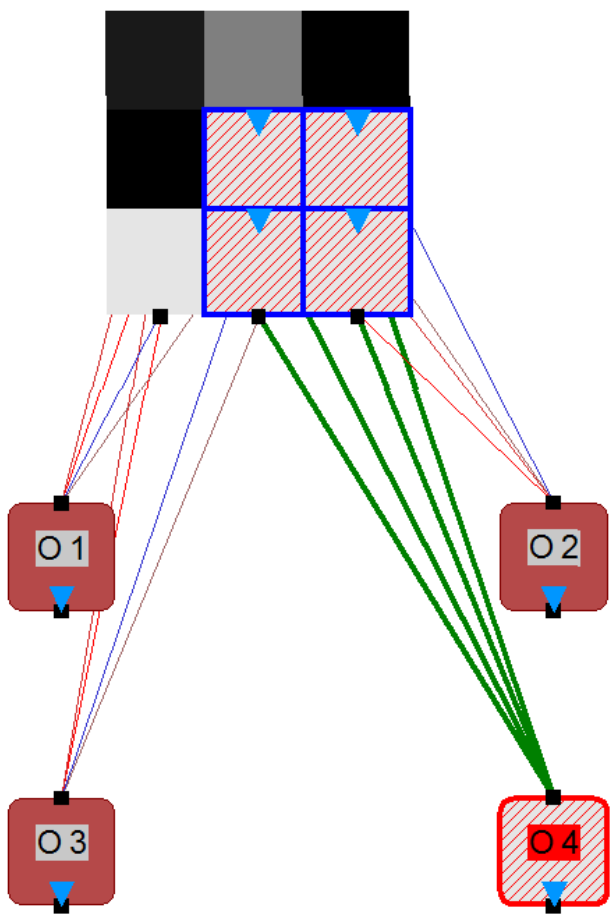
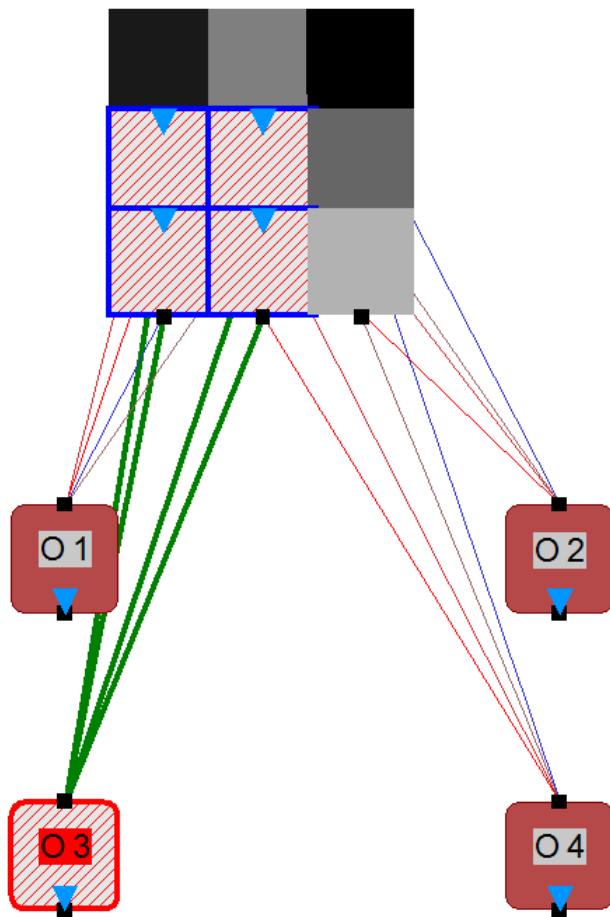
Part2



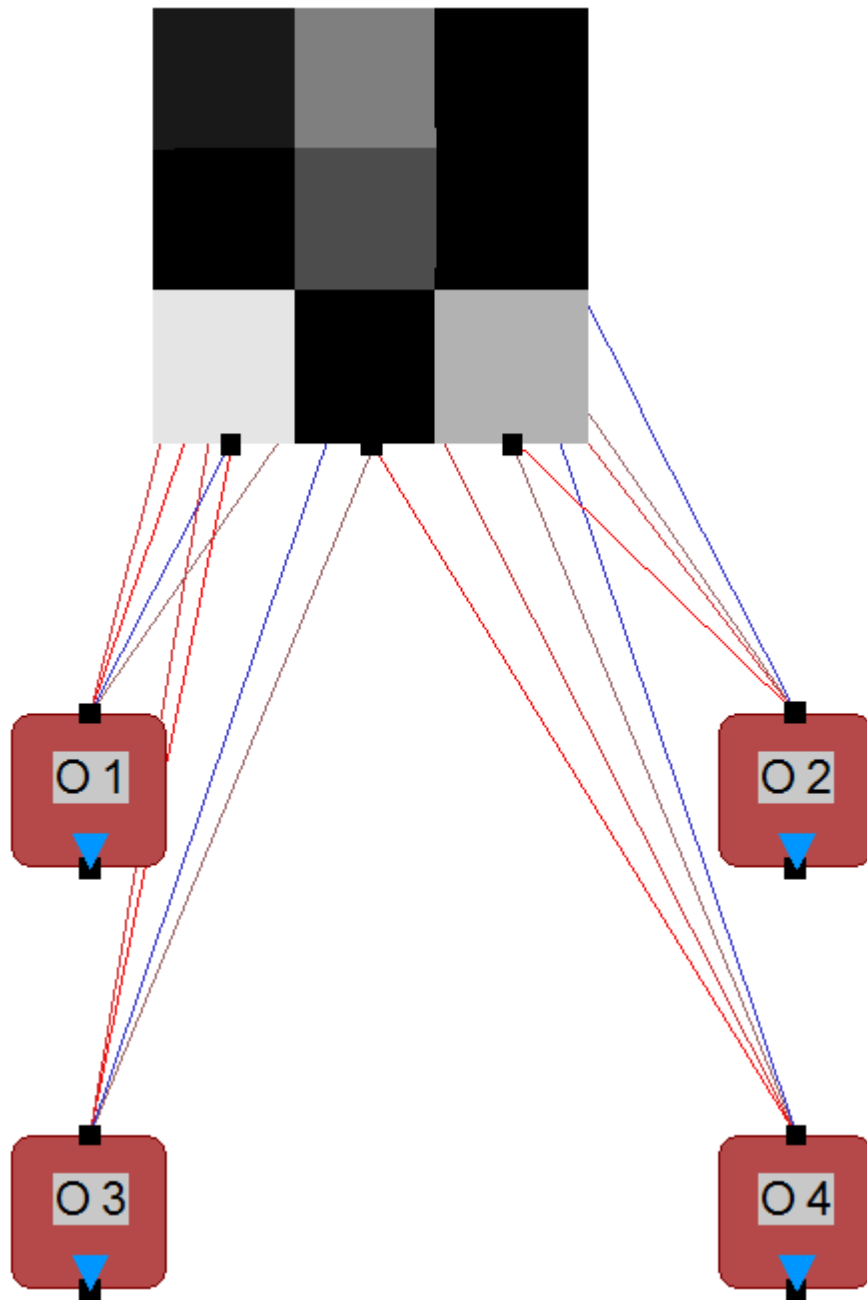


Part 3

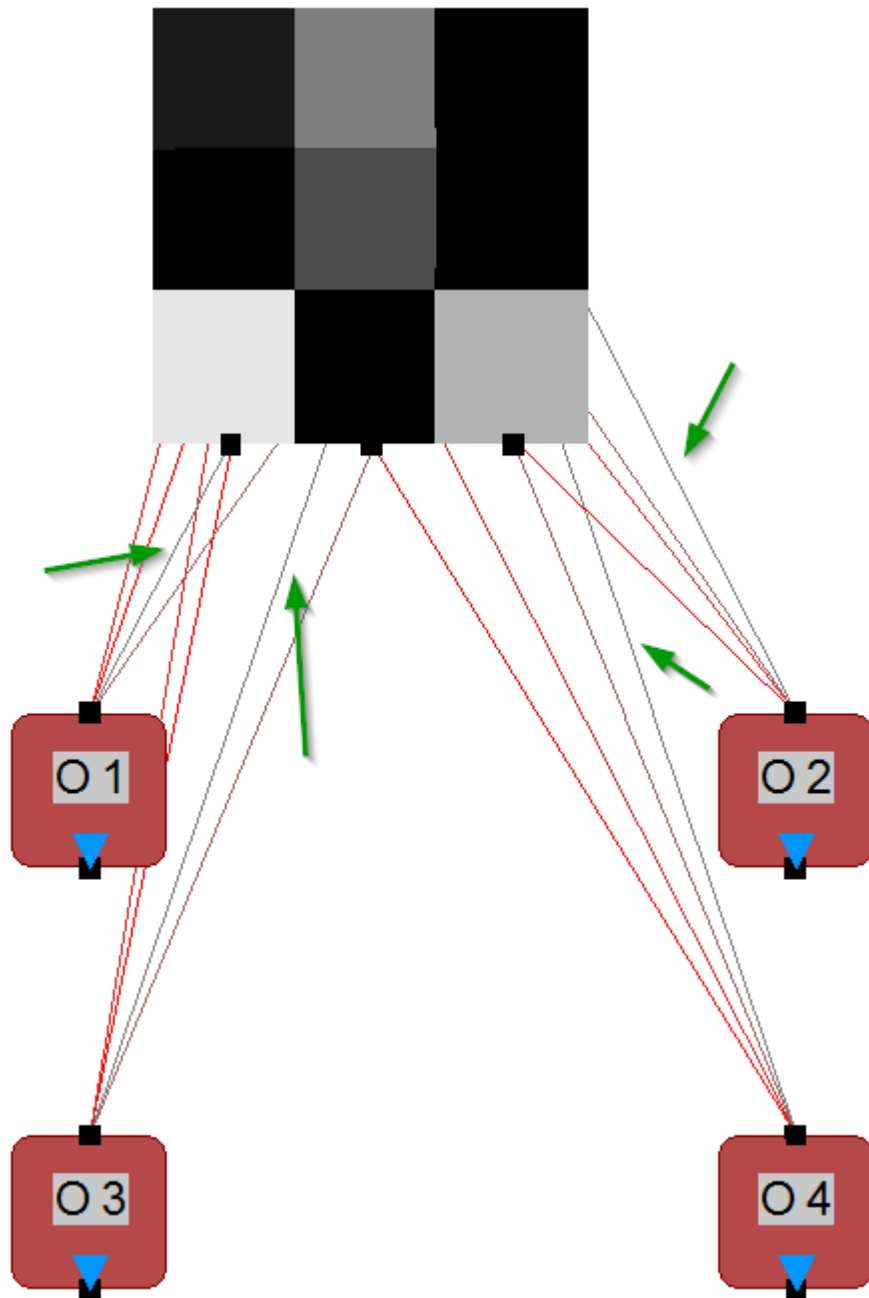
Part 4




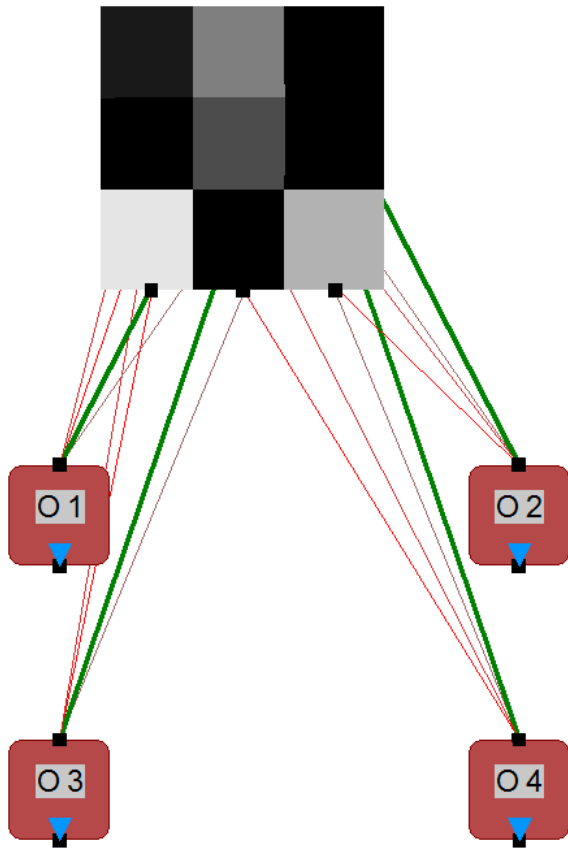
Without any selection we can see that the colors (representing the weight values) of the links show a specific pattern:



There are only four different colors (representing weight values) used. This is, because the connections to each of the output neurons use shared weights. I.e., each input array of 2 x 2 pixels is connected to an output neuron with the same weights as all other 2 x 2 arrays are connected to their related output neurons. If we double-click on one of the blue links, for instance, and change the (in this example negative) weight to 0 we will see that all four links follow this change. They all have become grey now:




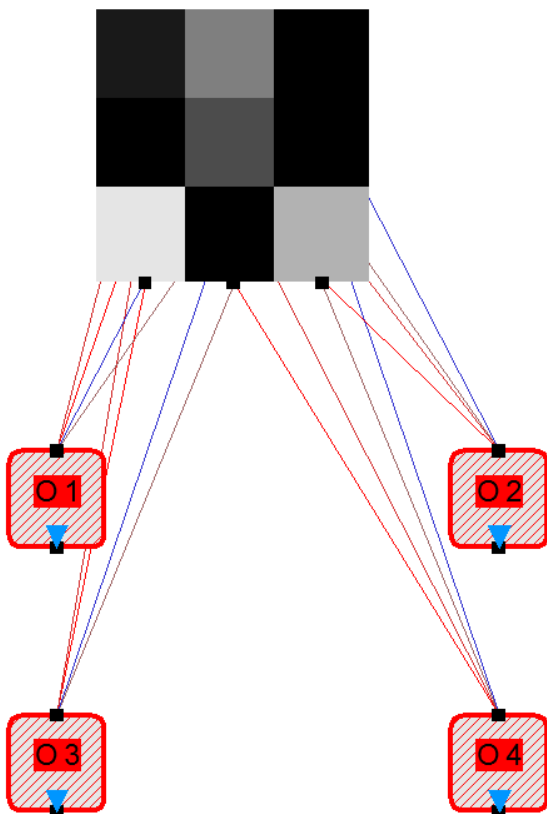
We can also visualize the convolution by selecting one of the links and then choose
 <Edit><Select...><Select Convolutions> or press the tool bar icon  :



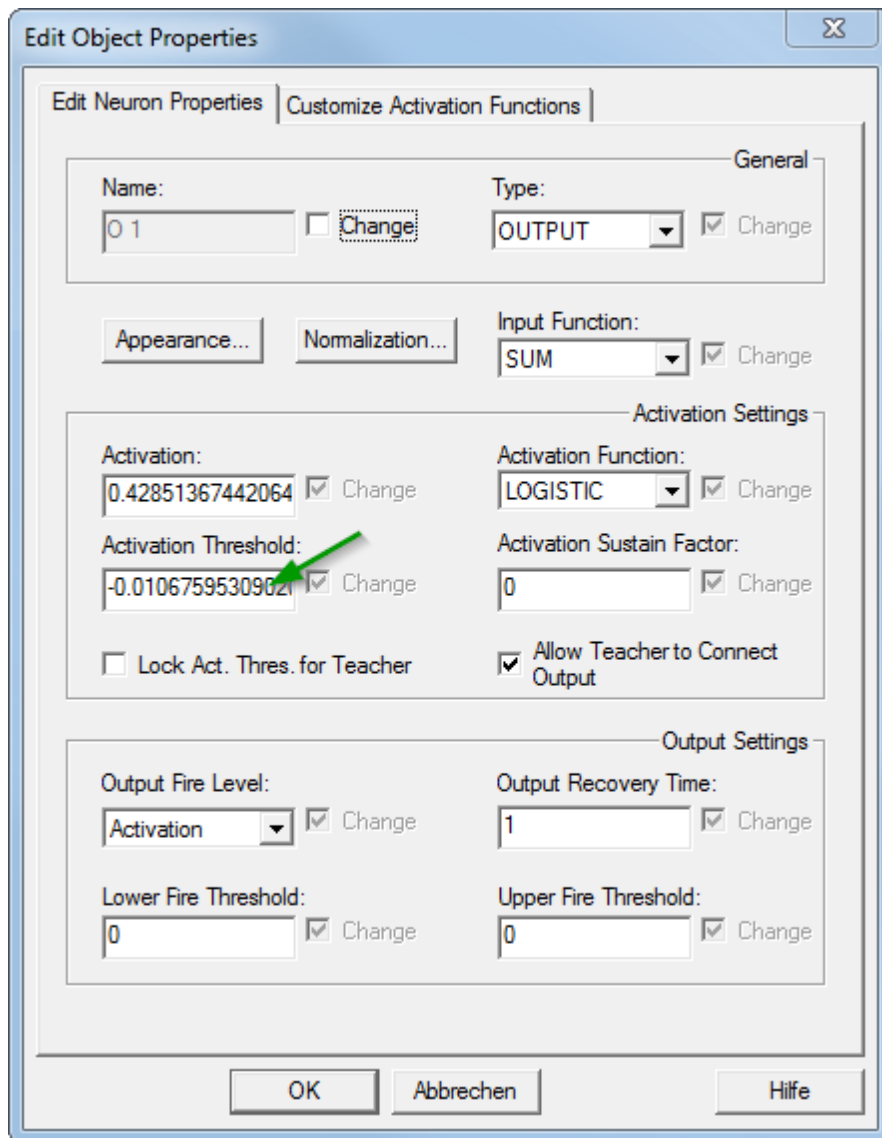
That's for the links.

The output neurons use convolution, too: Let's select one of the output neurons and then choose

<Edit><Select...><Select Convolutions> or press the tool bar icon  :



See that all output neurons have been selected, indicating that they all share one single threshold value. We can see this threshold value by double-clicking on one of the neurons:



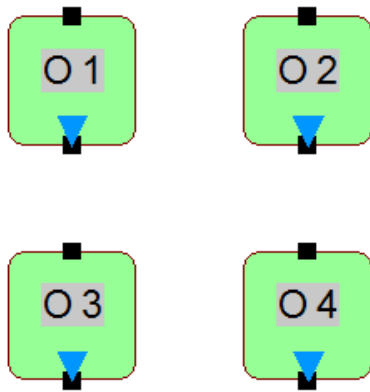
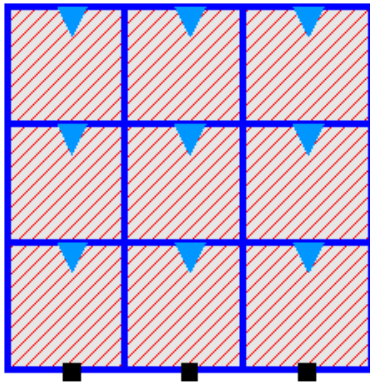
The next sections show how to establish and remove convolutions between neurons and links.


Convolutional Matrix Connection

Although convolutions between links can be established manually, the most convenient and practical way is to generate convolutional links by either using MemBrain's [Neuron Layer Insertion function](#) or by its matrix connection feature. The latter approach is explained here. It is best used when the neuron matrices to be interconnected are already existing. If the matrices have not yet been created then the faster approach is to use the [Neuron Layer Insertion function](#) instead.

Again, let's take the simple example here with 9 input pixels and 4 output neurons.

The following picture shows the net without any links yet. The 9 input pixels are [selected](#), the 4 output neurons have been [Extra Selected](#):



Now we choose <Edit><Connect...><Matrix Connect TO Extra Selection...> or press  on the tool bar or use the context menu <Matrix Connect TO Extra Selection...> that appears when right-clicking on one of the selected neurons. The following dialog comes up:

Matrix Connection Specification

Specify below how the sub groups shall be chosen to interconnect the current Selection and the current Extra Selection.

The connections will be arranged using rectangular neuron groups with the specified dimensions in neurons. Additionally you can specify if the groups shall overlap each other and if so, then how big the overlap area shall be.

Selection Grouping - Matrix dimensions = 3 x 3 neurons

Group X Size: Group Y Size: ☐ No Group Overlap
☒ Overlap Groups By: Neurons

The current settings result in 4 connection groups for this matrix

Connect TO:

Extra Selection Grouping - Matrix dimensions = 2 x 2 neurons

Group X Size: Group Y Size: ☒ No Group Overlap
☐ Overlap Groups By: Neurons

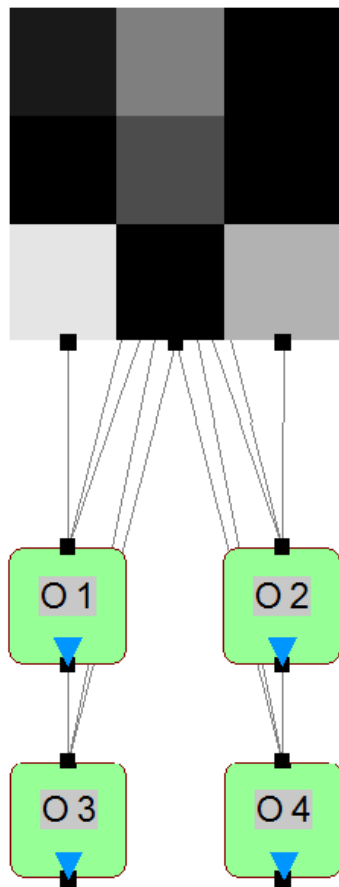
The current settings result in 4 connection groups for this matrix


☒ **Convolutional weights**

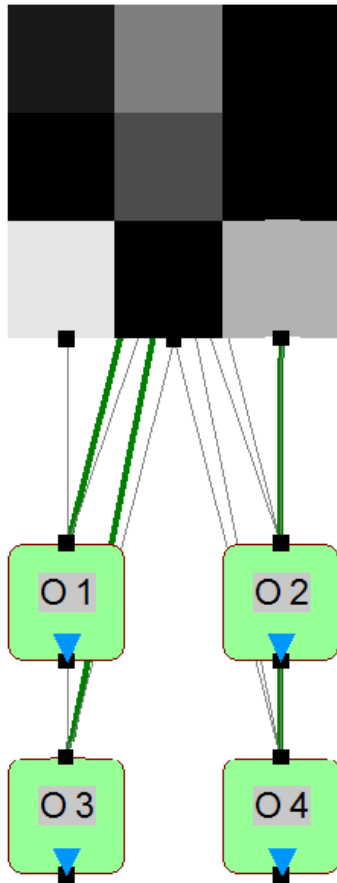
The matrix connection is feasible.

Cancel OK

We enter the values as indicated by the green arrows (don't forget ticking the box "Convolutional weights") and click <OK>. MemBrain now creates convolutional weights according to the specified matrices. The resulting net looks like this:




The created links are convolutional as you can easily find out by selecting one of the links and then using <Edit><Select...><Select Convolutions> or by pressing the tool bar icon  :




Making neuron layers convolutional

In order to make the neurons in a layer convolutional, simply:

- select all neurons in the layer
- right click on one of the selected neurons
- choose <Establish Convolution (Share Weight/Threshold)> from the context menu
Note that the command is also available via the menu category <Edit><Convolutions...> and also via the tool bar button 


Select Convolutions

In order to select all convolutions of the current selection, right-click on one of the selected objects, then choose <Select Convolutions (Objects with shared weights/thresholds)>.

Alternatively, use the same menu command in the menu category <Edit><Select...> or the toolbar button .

Remove convolutions

In order to remove all convolutions (i.e. weight/threshold sharing) of the current selection, right click on one of the selected objects, then choose <Remove Convolution(s) (Break Weight/Threshold Shares)>.

Alternatively, use the same menu command in the menu category <Edit><Convolutions...> or the toolbar button .

Neural Net Stock

MemBrain features a so called 'Neural Net Stock' which allows you to administer an arbitrarily large number

of neural nets in parallel while MemBrain is running.

The currently edited neural net can be captured on stock at any time during operation so that it can be re-loaded whenever desired. Capturing or loading a net on/from the stock is much faster than performing corresponding operations via files since the whole neural net stock is kept in RAM while MemBrain is running.

To manually capture the currently edited neural net on the stock select <Stock><Capture Current Net on Stock> from MemBrain's main window or click the toolbar icon:



The net will instantaneously be captured on the stock in background. Note that the status bar of MemBrain contains an indicator for the number of nets currently stored on the Stock. The number will increase with each click on the toolbar icon.

The most powerful feature, however, is the option to capture the best performing neural net during training automatically on MemBrain's neural net stock: Whenever the teacher detects a new minimum of the net error it updates the neural net copy on the stock to reflect the corresponding net. After the training has finished the net can be re-loaded from the backup copy on the stock. Thus, the risk of over training a neural net is resolved quite elegantly: If a net is trained using a training lesson and a validation lesson at the same time and the net error is adjusted to be the validation lesson then the net which performs best on the validation lesson is backed up on the stock automatically.

MemBrain also provides the capability of storing or loading the whole neural net stock to/from file. Using this feature whole sets of neural nets can be administered together in one project-like aggregation.

The neural net stock can be administered via the [neural net stock manager dialog](#) or via corresponding [script commands](#).

The neural net stock manager can be opened via the menu command <Stock><Stock Manager...> or via the toolbar icon



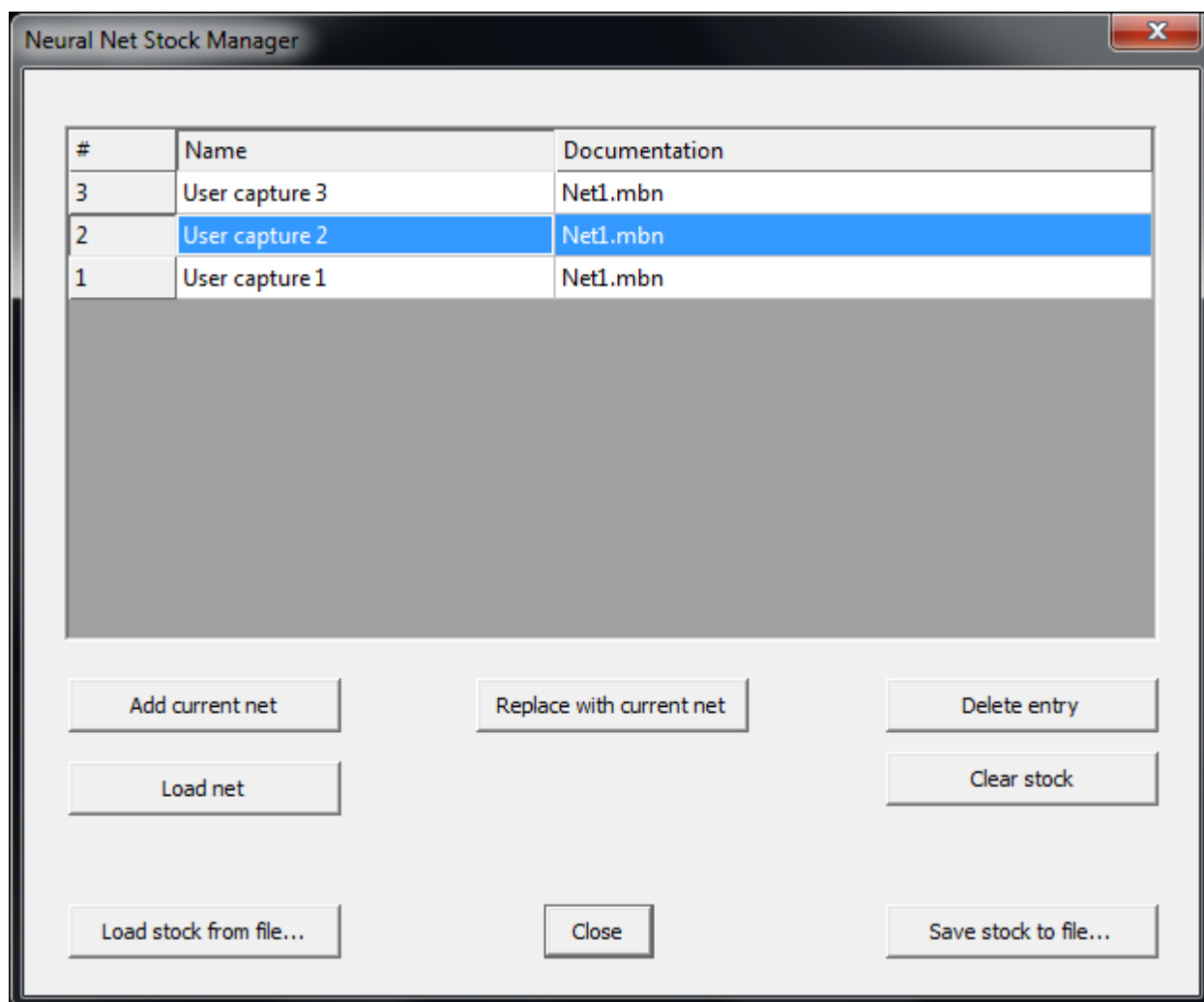
Click [here](#) for more help on MemBrain's neural net stock.

The Stock Manager

To open the stock manager select MemBrain's main menu entry <Stock><Stock Manager...> or click the following toolbar icon:



The stock manager will open and look something like this:



The list of nets always is presented in chronological order as the nets have been captured on the stock where the newest net is always on top of the list. The names and the documentation text for the nets can be edited by double clicking on the corresponding list fields.

The options available via the buttons on the dialog are as following:

- Add current net

A click on this button adds the currently edited neural net from MemBrain's main window to the stock. I.e., a new stock entry is created capturing the net in MemBrain's main window. This action can also be performed outside the stock manager dialog from MemBrain's main Window via menu (<Stock><Capture Current Net on Stock>) or via the toolbar icon



- Replace with current net

Similarly to the function 'Add current net' this function captures the currently edited net on the stock. However, it will not generate a new entry on the stock but replace the currently selected entry. Since this action cannot be undone MemBrain will ask before to perform the operation.

- Delete entry

The selected entry will be deleted from the stock. Since this action cannot be undone MemBrain will ask before to perform the operation.

- Load net

The selected net on the stock will be loaded to MemBrain's main window and replace the currently edited

net in the main window. The operation can be undone in MemBrain's main window via the normal Undo command in case it should have been used inadvertently.

- Clear stock

This command clears the whole stock (i.e. removed all nets from the stock). Since this action cannot be undone MemBrain will ask before to perform the operation.

- Load stock from file

The stock content will be replaced completely by another stock content loaded from file. The operation cannot be undone!

- Save stock to file

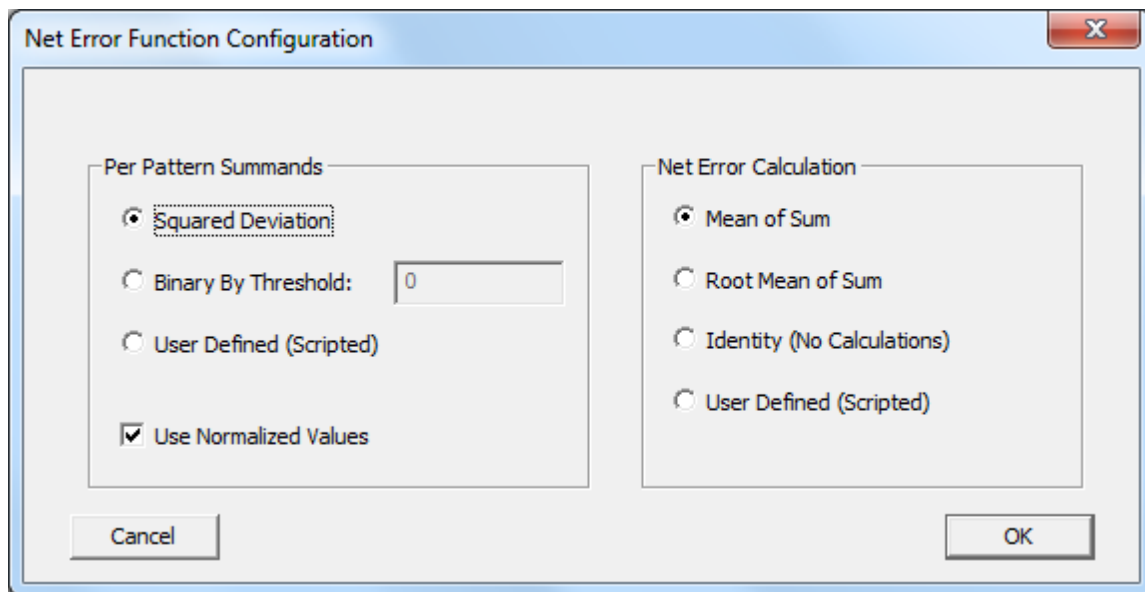
This option allows to save the whole stock to a file.

- Close

Closes the stock manager

Net Error Calculation

The way the net error is calculated is configurable through the menu item <Teach><Configure Net Error Function...> which brings up the following dialog:



Calculation of the net error is defined by two different algorithmic steps (left and right group above):

1. Calculation of the Per Pattern Summands. This operation is performed once for every pattern application and every output neuron of the net. It defines how the single summands are calculated which make up the error sum before the final net error calculation is applied to this sum.
2. The final Net Error Calculation which is applied on the sum after each Lesson run.

For step 1. above users can select from the following options:

- *Squared Deviation*

The summands are calculated from the squared deviation between target activation and actual activation of the output neuron.

- *Binary By Threshold*

If the absolute value of the deviation between target activation and actual activation of the output neuron

is bigger than the specified threshold, then the value of '1' is added as summand. Else nothing (value 0) is added as summand.

- *User Defined (Scripted)*

The calculation is a user defined script implementation as part of the currently registered [Scripted Elements](#).

If the check box *Use Normalized Values* is checked then MemBrain calculates the net error summands based on normalized values, i.e. based on the internally used activation function ranges of the neurons. If the box is unchecked then MemBrain uses the user defined activation ranges (i.e. the Normalization settings) of the neurons to calculate the net error.

For step 2. above users can select from the following options:

- *Mean of Sum*

The sum of all summands over each lesson run is divided by the number of summands. I.e. the net error will become the average summand value.

- *Root Mean of Sum*

Similar as 'Mean of Sum' but after calculating the average the square root function is additionally applied on the average.

- *Identity (No Calculations)*

No calculations are applied on the sum, i.e. the sum is used as net error

- *User Defined (Scripted)*

The calculation is a user defined script implementation as part of the currently registered [Scripted Elements](#).

The net error in MemBrain always is calculated on bases of the currently active **Net Error Lesson**. The active net error lesson is determined on the [Lesson Editor](#) and can be changed even during the teaching process.

Validating Your Net

In order to validate a net that has been trained using a supervised learning algorithm the net has to be confronted with data that has not been part of the data set used for the training of the net.

Thus, before starting to train your net you should ensure that you have both, training and validation data sets available, both of the same quality and structure.

Let's assume you have 1000 data pattern of good known data available in a MemBrain Lesson. Then you should, before starting to train your net, split off a certain percentage of the data into a separate validation Lesson. E.g. you could split off 15 % of the data. For time invariant nets the split off data should be selected by random to ensure a comparable data distribution over the training and the validation lesson.

You can use the [Lesson Editor](#) to automatically perform this split, see Lesson Editor's menu <File><Split Current Lesson...>.

When you have trained you should load your validation data into the Lesson Editor and get a quick impression on how your not trained validation data is interpreted by your net:

Select <Teach><Evaluate Net Error> or press <Ctrl> + U on the keypad. This will perform an (invisible) test run on the currently active Lesson in the Lesson Editor (which is your validation Lesson now) and update the net Error in the status bar of MemBrain and also in the [Error Graph](#) accordingly. This number now reflects the net error over the data in your validation Lesson. Additionally, the [Pattern Error Viewer](#) will be updated so that you can see the output reaction of your net for every pattern in detail together with the corresponding target data.

However, normally you will want to inspect the reaction of your net on the validation data in more details, e.g. using a spreadsheet program. For this purpose you can create a csv file with all results of your net on

the inputs of the validation data by a simple button click in the [Pattern Error Viewer](#).

You can also use MemBrain scripts that automate the process of training an validation for you. This way your own established MemBrain processes become automated and can be executed by a single button click. You will find example scripts [here](#) together with instructions on how to use them. Give it a try, it's really simple!

By the way: The same scripts can be used to employ your net to get prediction on new data. That's how you can use trained MemBrain nets to obtain results on unknown data.

Alternatively, you can use a validation lesson already during training. This allows to calculate your net error and visualize the net output on basis of a validation lesson 'On The Fly' already during training. See description of the [Lesson Editor](#) for more details.

Simple Validation Script

The original script that was posted here in former MemBrain help file versions has been massively improved and moved to the MemBrain User Forum for download.

Additionally, a script version for time variant nets used for time series prediction is provided.

Please follow the links below for downloading the latest versions of the scripts:

- For time invariant problems/networks:

English version of the user forum post:

[Universal Training and Validation Script](#)

German version of the user forum post:

[Universelles Trainings- und Validierungsscript](#)

- For time variant problems/networks (i.e. time series prediction):

English version of the user forum post:

[Universal Training and Validation Script including extrapolation of time series into the future](#)

German version of the user forum post:

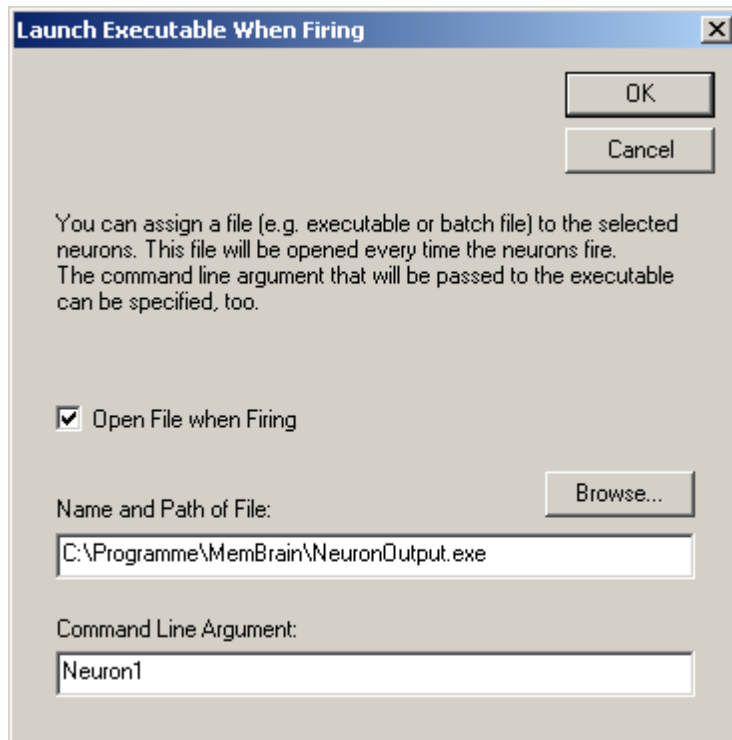
[Universelles Trainings- und Validierungsscript mit Extrapolation von Zeitreihen in die Zukunft](#)

Launch Executable by a Neuron

When a neuron fires it can optionally launch an executable or batch file. To be more precise, a file can be specified to be opened by Windows no matter what type is. It could also be a document of a type that is linked to certain application within Windows.

Together with the file to be opened you optionally can specify a command line argument that will be used when the file is opened. For example you could enter the neuron's name here so that the launched application would get informed about which neuron launched it.

To specify a file to be opened when a neuron fires, select one or more neurons and then choose <Extras><Executable When Firing...> from the main menu or from the context menu that appears when right clicking on one of the selected neurons with the mouse. The following dialog will show up.



Note the check box named <Open File when Firing>. If this box is checked the information below is relevant. Note that the box can also be grayed and checked. This means that some of the selected neuron have the option activated and some don't.

The path and filename to be opened as well as the command line argument are stored with every neuron separately. So you can specify a different file to be opened for every neuron.

Note that you can [display neuron fire indicators](#) in order to visualize when a neuron is firing.

Employ Trained Nets

In order to employ a trained net on new, unknown data, the same approach is used as for [validating](#) your net. You can use the same script as described in the chapter for validation or obtain results on new data manually through the Lesson Editor. Furthermore, you can use either the [MemBrain DLL](#) or the [source code generation feature](#) of MemBrain to employ trained nets in your own software.

Scripting

MemBrain supports a powerful C++/Java like scripting language. Most of MemBrain's functions can be used in scripts amongst other functions like starting external applications, reading or creating binary or ASCII files etc. MemBrain scripts can for example be used to:

1. Combine commands into script files and execute them in a single sequence from MemBrain's menu.
2. Create demos with MemBrain including user interaction through message boxes, file open/save dialogs etc.
3. Remote control MemBrain by providing and launching scripts from other software.
4. Train different nets or variants of a net in multiple ways and select the best candidate.
5. ...

The scripting language allows to control all core functions of MemBrain like for example:

- Loading and saving nets
- Loading, importing, saving and exporting lesson data

- Recording lesson data
- Teaching
- Thinking
- Configuring view

Besides this the scripting language supports auxiliary functions like

- Starting and terminating external applications
- Perform user interaction through message boxes, file selection dialogs, value input dialogs etc.
- Reading/Writing/Creating files e.g. to communicate with external applications.
- Communicate via serial ports
- ...

Note that MemBrain comes with its own [built-in source level script debugger](#) which streamlines the script development by using a comfortable editor including syntax highlight, break points, variable watch and many more features.

MemBrain's scripting language is based on [AngelScript](#) - a powerful scripting engine developed for high performance script execution. AngelScript scripts are not interpreted at run time but compiled into byte code and then executed on a virtual machine to maximize performance. Still, all you need to run AngelScripts for MemBrain is readily included in MemBrain so you don't have to worry about any installations or external development tools. MemBrain takes care of the whole compilation and execution process.

AngelScript's syntax is very close to C++ or Java and thus provides powerful yet simple to learn syntax combined with a strong type checking and an object oriented approach.

The [following chapters](#) describe the scripting language and the scripting functionality of MemBrain in detail. However, although AngelScript is fully integrated in MemBrain it is a third party scripting engine for which separate documentation is provided at <http://www.angelcode.com/angelscript/>. A copy of the documentation at the time point of this MemBrain release has been copied to your local drive during installation of MemBrain and is available in the sub directory 'AngelScript\docs' of your MemBrain installation (file 'Index.html').

Thus, this help file will not describe the AngelScript syntax as far as the scripting language itself is concerned but only describe the MemBrain-related aspects of the scripting implementation, i.e. the MemBrain-specific elements available in scripts and the steps needed to run AngelScripts in MemBrain.

General Description

MemBrain's scripting language is based on the full scripting syntax provided by [AngelScript](#).

MemBrain also incorporates all so called 'Add-Ons' provided by the Script Language Author.

Every script requires the function

```
void main()
{
    ...
}
```

to be implemented. This is the script entry point, i.e. the main function of the script where execution begins. When the script execution returns from this function the script exits automatically, i.e. is finished. Certainly an arbitrary number of other sub functions can be implemented and called in a script.

MemBrain scripts can be spread over several files by use of the **#include** directive. With #include other script files can be added to the current script. The behaviour is the same as if all the code would be placed in one script file.

Before a script is executed it is automatically checked for syntactical correctness and compiled into byte code that can be executed by the AngelScript virtual machine that is integrated in MemBrain. Certainly that doesn't imply that a certain script can be executed without errors as there might be also semantic errors in the script or data that is loaded by the script may be not in the correct format for the script commands.

If you want to check the syntactical correctness of a script without executing it you can choose **<Scripting><Check Script Syntax...>** from MemBrain's main menu.

After every script execution a script result file 'scrend.txt' will be generated which can be used to process error messages that occurred during script compilation or execution. For example many text editors can be set up to extract information about error locations in a file (line and columns) using regular expressions. These editors can use the 'scrend.txt' file to allow the user to directly jump to the location of an error within a script for example. More details about the scrend.txt file can be found [here](#).

Errors during script execution are handled differently depending on the type of error and the called function. Some errors lead to the script being aborted immediately. Error messages are then reported through the script execution result file and are additionally displayed to the user in the scripting trace window which will automatically open in such situations. This kind of error handling applies to all script commands that modify data within MemBrain, e.g. when a lesson shall be loaded into the Lesson Editor or when a certain pattern within a lesson shall be set as the active one.

Other errors that don't impact the further operation of MemBrain are signalled through script function return values and the appropriate error handling is left to the script. An example for this is setting the displayed range in the Pattern Error Viewer. The corresponding script function will return <false> in case the given range is invalid.

For details on how specific errors are handled during script execution see the corresponding function descriptions in the [Command Reference](#) section.

Note that during script execution MemBrain's user interface is not blocked, i.e. you can interact with the script by performing operations in parallel. Nevertheless this may lead to script abortion if errors occur due to the interaction. One possibility of useful interaction with a script is to start teaching at some point in a script and waiting for the teach procedure to be finished. The user can then stop the teacher and the script will automatically continue to run e.g. to validate the trained net against some test data in a lesson and to record the results in another lesson.

Note that a teacher always stops automatically if it's target net error is reached. Thus, a script can also wait until the target net error is reached by waiting on the teacher to finish.

See [here](#) on how to start scripts manually. If you want to read more about remote scripting, click [here](#).

If you want to read more about the general syntax rules for MemBrain scripts then click [here](#).

For a reference on the available script commands and elements along with their their syntax and parameters, see [here](#).

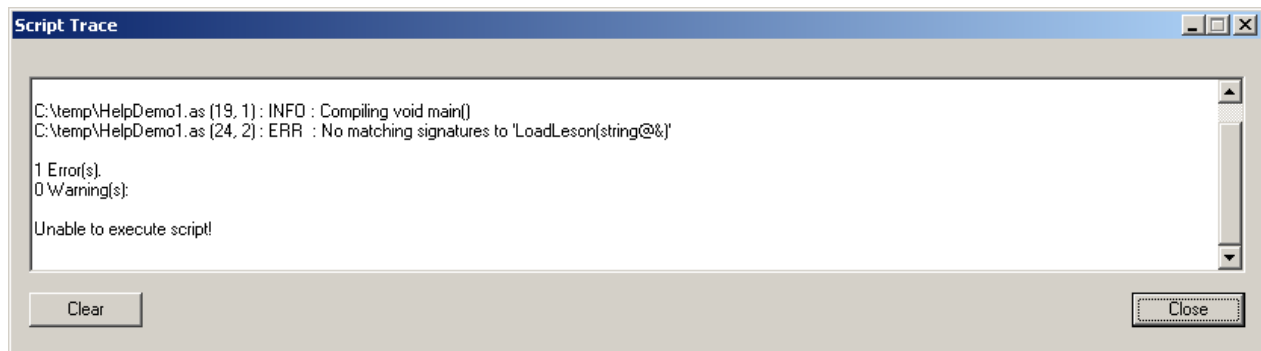
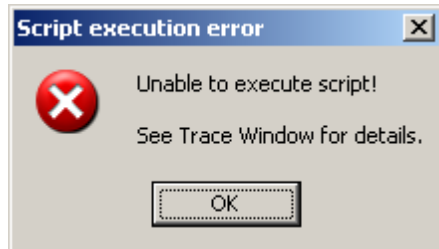
The preferred way of developing MemBrain scripts is the [MemBrain Script Debugger](#). However, other options are available, too:

Since the MemBrain scripting language is very similar to C++, C# or Java you can use a text editor that supports syntax highlighting for C++, C# or Java for editing MemBrain scripts. Configure the file type *.as to be displayed like these files in the text editor. This will improve the readability of the scripts significantly! Also, since the file [scrend.txt](#) is always created after every script execution, no matter whether successful or not, this file can be used to parse compiler and execution errors and use this information to jump to the corresponding script line inside a text editor. MemBrain can be registered as compiler for script files as part of the external tool setup of many text editors. The script that shall be compiled and executed can be provided to MemBrain through the [command line](#).

Executing Scripts Manually

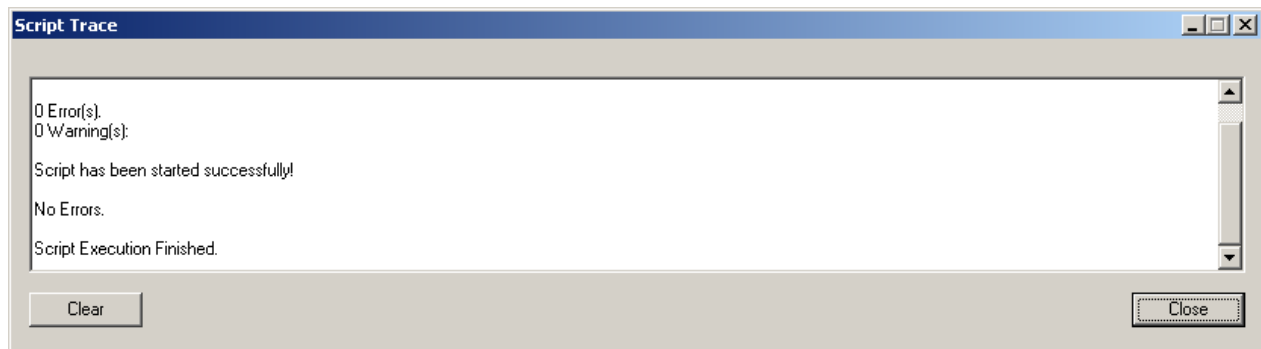
If you want to execute a script manually from within MemBrain choose **<Scripting><Execute Script...>** from MemBrain's main menu.

A dialog will open up to select the script you want to execute. The script will automatically be routed through a syntax check before execution i.e. if the script contains syntactical errors you will be informed and none of the commands in the script will be executed. In this case you will get an error message that states the reason for the problem. The scripting trace window will contain details on the source of the error(s):

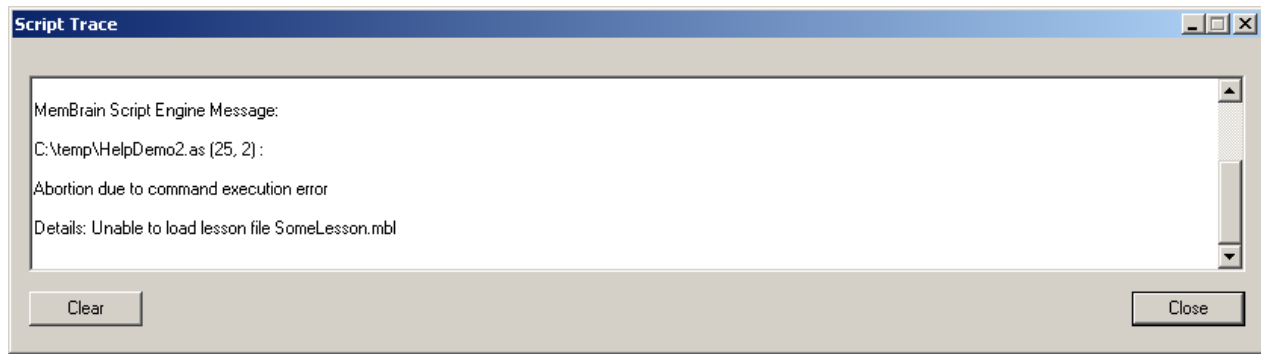


In the above example the script syntax pre-check failed for the given script in line 24, column 2 because the function call 'LoadLeson(...)' could not be resolved. Reason is a missing letter: The correct function name is 'LoadLesson(...)'.

If the script pre-check passes then the execution of the script starts which will be stated as a message in the scripting trace window. After successful execution of a script you will be informed by an additional message in the scripting trace window as indicated below.



An error during script execution leads to automatic abortion of the script. You will be informed about this by a corresponding message in the scripting trace window, e.g. as following. The error in this case occurred while executing line 25 of the script file because a MemBrain lesson file could not be loaded.



Note that you can always re-execute the last executed script by selecting the toolbar button



It is possible to change the script selection for execution using the corresponding drop down combo box in the Scripting tool bar:

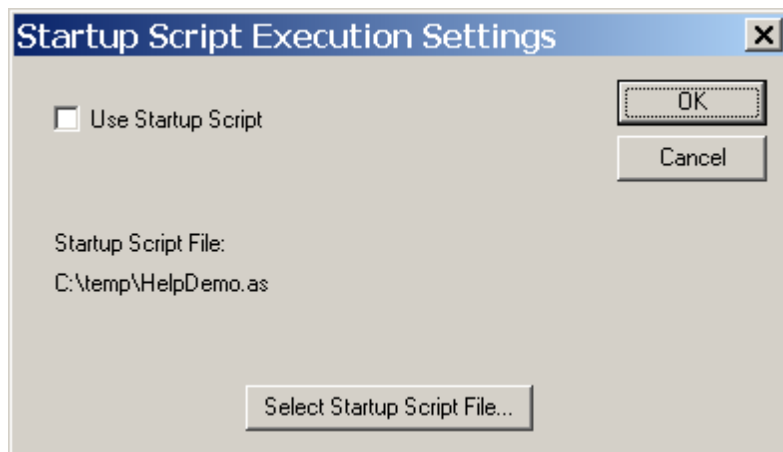


The list keeps the history of the last 30 executed or compiled scripts so that they can be quickly selected for re-execution.

The Startup Script File

MemBrain can be equipped with a script file that is automatically executed after startup of MemBrain.

Select **<Scripting><Use Startup Script...>** to activate/deactivate this option and to select a corresponding script file. The following dialog will appear.



If the option is activated (check box in the upper left corner of the dialog) then the given script file will be executed every time when MemBrain is started up.

To select or change the script file that shall be executed click the button on the the lower border of the dialog. The current selection will be printed above the button as shown in the example dialog above.

If activated then also a check mark will be placed beneath the named menu option in MemBrain's **<Scripting>** menu.

If a net is given to be loaded by the command line of MemBrain then first the net is loaded and then the

Startup script is executed.

Note: If a [command line script](#) is given then MemBrain will not execute the Startup script file even if activated, i.e. the command line script will be executed instead of the Startup script.

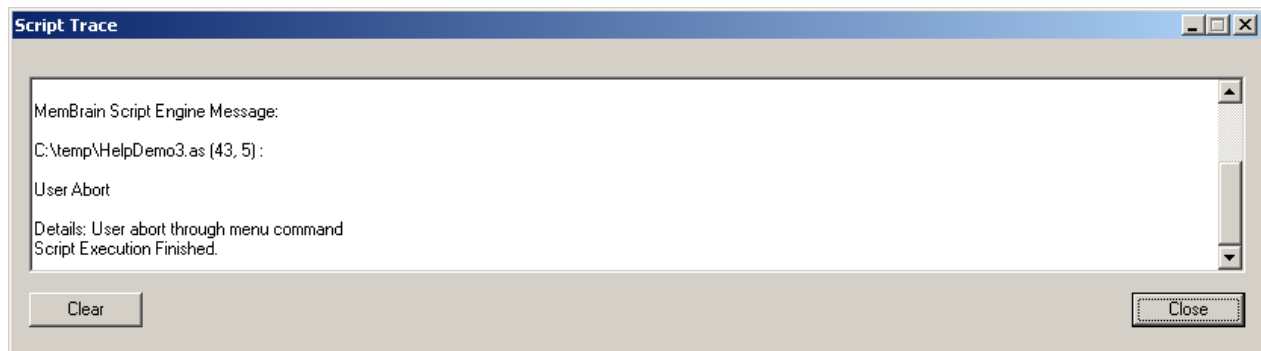
Aborting or Suspending Script Execution

You can abort a script execution at any time using MemBrain's menu command <Scripting><Abort Script Execution> or by clicking on the toolbar symbol



You will be prompted to confirm the abortion of a script.

If the script was started manually you will be prompted in the scripting trace window as following once the script has been aborted.



If you want to temporarily suspend a script using the toolbar button



or by selecting the corresponding command from the <Scripting> menu. To resume the script execution press the button again. The script will continue to run at the next statement following the point of suspending. Note that if the last executed script command was a Sleep command then execution will be resumed with the command following the sleep command, i.e. the sleep operation itself will not be resumed!

MemBrain also provides a function prototype that you can implement in your scripts and that will automatically be called when a MemBrain script is aborted (for any reason):

| | |
|--|---|
| Function Prototype to handle Script Abortion | void OnAbortScript() |
| Purpose | Function that is automatically called by MemBrain whenever a script aborts, no matter for what reason. Implementation of this function is optional. |
| Parameters | None |

| | |
|----------|---|
| Comments | <p>A good example for an implementation of this function is to stop any ongoing teach or think process:</p> <pre>void OnAbortScript() { StopTeaching(); StopThink(); }</pre> |
|----------|---|

Remote Script Execution

Scripts in MemBrain can also be executed remote controlled through a simple file interface:

MemBrain can be enabled to cyclically poll for a so called remote script start file that is a simple text file with the fixed name **scrstart.txt**. The remote script start file must contain the name of the script file that shall be executed as the first line that contains text. Results of the script execution will be reported in another file with fixed name, the file **scrend.txt**.

If MemBrain's remote scripting functionality is enabled it will read in the script file name from the scrstart.txt file, delete the scrstart.txt file, delete the scrend.txt file and try to execute the script. After execution, no matter if successful or not, a new version of scrend.txt will be written which contains the overall result of the script execution. During remote script execution no script execution related errors will be announced to the user. Instead of showing a message box the script execution will be aborted and the scrend.txt file written. This allows external software to be informed about errors that occurred during script execution without MemBrain waiting for user input.

If script execution runs without errors the scrend.txt file will contain the following:

OK

Start: 09 Nov 2006, 08:28:20

End: 09 Nov 2006, 08:28:26

where Start respectively End depict the start and end time point of the script execution.

Upon errors during script pre-check or execution the file scrend.txt will contain something like the following:

Error

Start: 16 Feb 2009, 17:52:45

End: 16 Feb 2009, 17:52:45

MemBrain Script Engine Message:

C:\temp\HelpDemo.as :

Error(s) during compilation!

C:\temp\HelpDemo.as (19, 1) : INFO : Compiling void main()

C:\temp\HelpDemo.as (29, 15) : ERR : No matching signatures to 'GetLessonOutputCounts()'

C:\temp\HelpDemo.as (54, 2) : WARN : Unreachable code

The message upon an error certainly will vary but the first line will always contain the text "Error".

The path where the remote scripting files are processed can be [set by the user](#).

Note1: If you don't use remote scripting you should keep the functionality disabled as MemBrain looks for a

remote scripting file appr. every 0.5 seconds when the option is enabled which puts additional load on your computer.

Note2: The file 'scrend.txt' is always generated, even if the script was executed manually. The content of this file can also be used to directly navigate to locations of script file errors for example since the error messages always contain the line and column in the code where an error occurred. Many text editors support jumping to a certain location in a file based upon regular expressions that evaluate information from another file.

Set Path for Remote Scripting

You can specify the **path** where MemBrain shall look for the **scrstart.txt** file for remote script execution and where the **scrend.txt** file shall be placed after every script execution:

Choose <Scripting><Remote Scripting Path...> and select a path.

MemBrain will store this setting in its configuration so you have to set this path only once.

The script file name contained in the scrstart.txt file (specifying the script that shall be executed, see [here](#)) is always interpreted relative to this path. So, if you just specify the file name of the script without any path information then

MemBrain will look for the script file in the remote scripting path.

Nevertheless you can also specify the script file with absolute path information in scrstart.txt, e.g. C:\Scripts\MyScript.txt.

It is also possible to specify the script file using relative path information like for example ..\Scripts\MyScript.txt which will go one level up from the remote scripting path and then down again into the subdirectory *Scripts* which in this case has to be on the same level as the remote scripting path. Note that the remote script output file *scrend.txt* will be placed in the remote scripting path independent from the location of the executed script.

Command Line Script Execution

Scripts can be also executed through the command line when starting MemBrain:

In the command line add

/S <Script Name>

where <Script Name> represents the path and name of the script to be executed.

A script on the command line has to be placed in the last position of the command line.

E.g. the command line:

MemBrain C:\MyNets\MyNet.mbn /S C:\MyScripts\MyScript.txt

first loads the net 'MyNet.mbn' and then executes the script 'MyScript.txt' in the given directories.

Note that the file names also may contain spaces. The space between /S and the script name is optional, too. Even the space before the /S can be omitted. Detection is not case sensitive i.e. /s and /S both act the same way.

Command line scripts will be executed in the same way as [remotely started scripts](#) i.e. there will be no message boxes shown upon errors or after script execution but the remote script result file **scrend.txt** will be updated with the result of the script execution as described in the corresponding section.

Note: If a command line script is given then MemBrain will not execute the Startup script file even if activated, i.e. the command line script will be executed instead of the Startup script.

Command Line Script Compilation

Scripts can be also compiled through the command line when starting MemBrain:

In the command line add

/C <Script Name>

where <Script Name> represents the path and name of the script to be compiled.

Note that the file names also may contain spaces. The space between /C and the script name is optional, too. Even the space before the /C can be omitted. Detection is not case sensitive i.e. /c and /C both act the same way.

Command line scripts will be compiled in the same way as [remotely started scripts](#) i.e. there will be no message boxes shown upon errors or after script compilation but the remote script result file [scrend.txt](#) will be updated with the result of the script compilation as described in the corresponding section. MemBrain will automatically terminate after compilation is complete, i.e. the command line option /C allows to use MemBrain as a command line compiler that can be invoked by external tools like editors for example.

Script Syntax

The MemBrain scripting language is realized using the [AngelScript](#) scripting engine. Thus, the basic syntax is defined by the AngelScript syntax which is in many ways identical to C++. For a detailed description on the scripting language please refer to the corresponding documentation which is available on the AngelScript homepage, using the link given above.

A snapshot of the documentation has been copied to your MemBrain installation directory and can be accessed by clicking [here](#).

For a quick start scripting samples are available on the MemBrain homepage as a first reference for users. Still, these examples only use a fairly small subset of all the AngelScript features so there is a lot more to discover here!

MemBrain scripts are written in simple text files. The default extension for a script is *.as although MemBrain also accepts different file extensions.

The preferred way of developing MemBrain scripts is the [MemBrain Script Debugger](#).

MemBrain publishes many functions that are directly accessible from within the scripts. This means that MemBrain provides a scripting engine with powerful syntax and performance combined with all you need to control MemBrain's core functions from within the scripts. See [here](#) for a full list of all available script commands and objects and how to use them.

Every MemBrain script has to consist of one single file. Please note that the entry point for every MemBrain script is the function

```
void main()
{
    ...
}
```

which has to be implemented in every script. The script automatically exits once the script returns from this function. Scripts certainly may contain an arbitrary number of other functions which can be called within the scripts just like in any C++ program.

Also, when you are completely new to C++ or Java please keep in mind that these languages are case sensitive throughout. Thus, for example 'LoadLesson(...)' is a valid MemBrain function name while 'loadLesson(...)' will be unknown to the MemBrain script compiler. The same case sensitive interpretation

certainly applies to variables and AngelScript keywords.

Note:

You can pre-check the syntax of a complete script without executing it using MemBrain's menu command **<Scripting><Check Script Syntax...>**.

File Name parameters in Scripts

When file names are used as parameters in MemBrain scripts these are always interpreted relative to the location of the script. File names can be specified without path information, with relative path information or with absolute path information.

Some examples show how path information can be used in scripts:

1. Specifying file without path information

```
OpenNet("MyNet.mbn");
```

Opens a MemBrain network file located in the same directory as the executed script.

2. Specifying file with relative path information

```
OpenNet("BrainsMyNet.mbn");
```

Opens a MemBrain network file located in the sub directory *Brains* which is located below the directory of the executed script.

```
OpenNet("../BrainsMyNet.mbn");
```

Opens a MemBrain network file located in the directory *Brains* which is located on the same level as (parallel to) the directory of the executed script.

3. Specifying file with absolute path information

```
OpenNet("C:\BrainsMyNet.mbn");
```

Opens a MemBrain network file located as specified independent from the location of the executed script.

Scripted Elements

In addition to stand-alone executable scripts MemBrain features so called 'Scripted Elements'.

These Scripted Elements are optional scripted overrides of selected MemBrain internal functions. I.e. the user can modify the behaviour of certain MemBrain elements through implementing predefined script functions and activating these Scripted Elements instead of the built-in function versions.

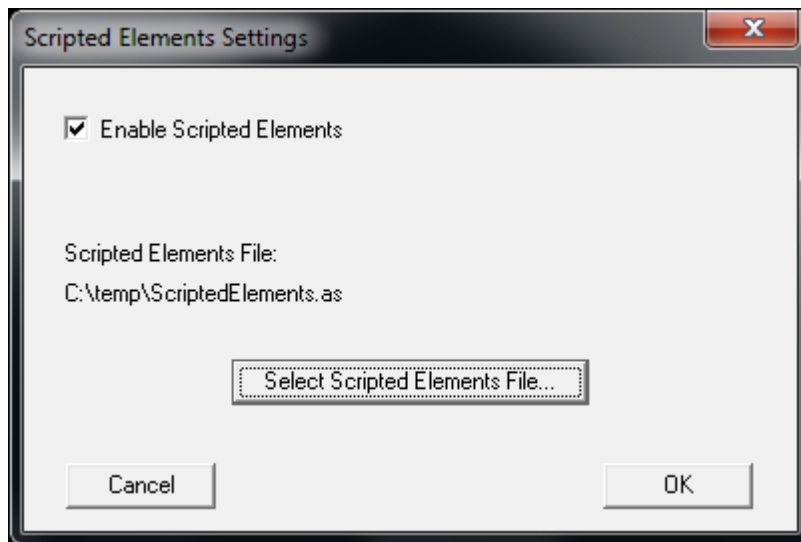
Currently MemBrain features the following Scripted Elements:

| | | |
|------------------|---|---|
| Scripted Element | double CalculateNetErrorSummand(double activation, double targetActivation, uint outNeuronNum) | |
| Purpose | Function to calculate the Net Error Summands as part of the Net Error Calculation | |
| Parameters | activation | The actual activation of the output neuron in reaction to the currently applied input pattern |

| | | |
|----------|--|--|
| | targetActivation | The desired target activation for the output neuron as per the definition of the currently applied input pattern |
| | outNeuronNum | Number of the currently processed output neuron starting with '1' Note: This parameter is of informational use and can probably be ignored for most of the typical function implementations |
| Comments | <p>If implemented and activated this function is called during the net error calculations (e.g. during teaching) once for each pattern application and each output neuron. It must return the desired value for the net error summand that contributes to the net error calculation.</p> <p>For more details on how to activate the use of this function see here.</p> | |

| | | |
|------------------|--|--|
| Scripted Element | double CalculateNetError(double errorSum, uint summandCount) | |
| Purpose | Function to calculate the resulting Net Error from the accumulated sum after each lesson run as part of the Net Error Calculation | |
| Parameters | errorSum | The accumulated sum for the net error calculation during the just performed lesson run. |
| | summandCount | The number of summands that have been incorporated into the sum during the last lesson run |
| Comments | <p>If implemented and activated this function is called once after each lesson run in order to finalize the net error calculation based on the accumulated sum and the number of summands. The function must return the resulting Net Error value.</p> <p>For more details on how to activate the use of this function see here.</p> | |

In order to configure MemBrain to either use or ignore Scripted Elements select <Scripting><Scripted Elements...> which will bring up the following dialog.



With the checkbox in the top of the window the use of Scripted Elements in MemBrain can be enabled or disabled in general. When enabled you can select a script file that contains your Scripted Elements implementations by a click on the corresponding button.

Confirmation with <OK> will compile your script file and end the dialog if successful. If unsuccessful you will be prompted and the Script Trace window with corresponding script compiler error messages will pop up automatically.

For implementation of your scripted elements you can use the normal syntactic feature set of the scripting language. However, the compiler won't accept any commands that would interact with MemBrain, i.e. most MemBrain specific script commands are not accepted as part of the Scripted Elements.

Command Reference

This chapter lists all available **MemBrain specific commands of the MemBrain scripting language** according to the following sub-categories.

- [MemBrain Application](#)
- [Handling Neural Nets](#)
- [Handling Lesson Data](#)
- [Teaching](#)
- [Thinking](#)
- [Adjusting the View](#)
- [Controlling the Weblink](#)
- [Communication with the user](#)
- [Controlling External Applications](#)
- [Stock Management](#)

- [Arbitrary file Access](#)
- [Strings](#)
- [String Utilities](#)
- [Maths](#)
- [FFT Calculations](#)
- [Serial Ports](#)

Note that many of the objects and methods listed below are part of the script add-ons delivered by AngelScript and not all of these add-ons are fully described below. As of writing, MemBrain supports all add-ons which are documented here:

Script extensions

- **string object**
- **array template object**
- **any object**
- **ref object**
- **weakref object**
- **dictionary object**
- ~~**file object**~~
- **filesystem object**
- **math functions**
- **grid template object**
- **datetime object**

Note that instead of the AngelScript file object add-on MemBrain provides its own file class as documented above in section [Arbitrary file Access](#).

See http://www.angelcode.com/angelscript/sdk/docs/manual/doc_addon_script.html for more details on the add-ons.

MemBrain Application

All script commands addressing the MemBrain application itself are listed in this section.

| | | |
|--------------|---|---|
| Command | void GetVersionInfo(string& out versionInfo) | |
| Purpose | Get the MemBrain version information string. | |
| Parameters | versionInfo | string variable receiving the version information |
| Return value | none | |
| Comments | none | |

| | |
|----------|--------------------------------|
| Function | void Sleep(uint timeMs) |
|----------|--------------------------------|

| | | |
|------------|--|--|
| Purpose | Pause the script execution for some fixed time | |
| Parameters | timeMs | Duration of sleep state in units of ms |
| Comments | <p>The given sleep time is a minimum value. Depending on the workload on the computer, script processing may be further delayed until processing resources get free.</p> <p>Note 1: A teacher always stops automatically if it's target net error is reached. Thus a script can also wait until the target net error is reached by waiting on the teach process to finish.</p> <p>Note 2: You can interact with a script through the user interface of MemBrain: If you wait for a think process to be finished in a script the script will continue automatically once you stop the think process over the user interface. This certainly also applies for the teach process.</p> | |

| | | |
|------------|---|--|
| Function | void SleepExec() | |
| Purpose | Pause the script execution until the currently active teach or think procedure is finished | |
| Parameters | None | |
| Comments | <p>Note 1: A teacher always stops automatically if it's target net error is reached. Thus a script can also wait until the target net error is reached by waiting on the teach process to finish.</p> <p>Note 2: You can interact with a script through the user interface of MemBrain: If you wait for a think process to be finished in a script the script will continue automatically once you stop the think process over the user interface. This certainly also applies for the teach process.</p> | |

| | | |
|--------------|--|--|
| Function | bool SleepExec(uint maxTimeMs) | |
| Purpose | Pause the script execution until the currently active teach or think procedure is finished or until the given maximum time has elapsed, whatever happens first. | |
| Parameters | maxTimeMs | Maximum time to wait for the teach or think process to finish. |
| Return value | true if end of teach or think process has caused the function to return. false if reason for return was timeout. | |
| Comments | <p>Note 1: A teacher always stops automatically if it's target net error is reached. Thus a script can also wait until the target net error is reached by waiting on the teach process to finish.</p> <p>Note 2: You can interact with a script through the user interface of MemBrain: If you wait for a think process to be finished in a script the</p> | |

| | |
|--|--|
| | script will continue automatically once you stop the think process over the user interface. This certainly also applies for the teach process. |
|--|--|

| | | |
|------------|--|---|
| Function | void StartTimer(uint timeMs) | |
| Purpose | Start a one shot timer with the given elapse time | |
| Parameters | timeMs | Duration until timer elapses in units of ms |
| Comments | Use function IsTimerExpired() to check if the time has elapsed since start of the timer. When the function is called while the timer is already running then it is re-started with the given time value, i.e. the previous time value is discarded. | |

| | | |
|--------------|---|--|
| Function | bool IsTimerExpired() | |
| Purpose | Check if the timer has expired. | |
| Parameters | none | |
| Return value | true if timer has expired. | |
| Comments | Note that when the timer has not been started at all the function still returns true. | |

| | | |
|--------------|---|--|
| Function | bool IsTeaching() | |
| Purpose | Check if MemBrain is currently running a Teach process. | |
| Parameters | none | |
| Return value | true if teaching is in progress. | |
| Comments | When using this function to wait for the end of the teaching in some loop then the loop should also contain Sleep(...) statements that free processing resources for the system. Otherwise the script will unnecessarily consume processing time while waiting for the teach process to end. If you don't intend to perform any other script operations during waiting for the teach process to end then better use SleepExec(...) instead. | |

| | | |
|--------------|---|--|
| Function | bool IsThinking() | |
| Purpose | Check if MemBrain is currently running a Think process. | |
| Parameters | none | |
| Return value | true if thinking is in progress. | |
| Comments | When using this function to wait for the end of the thinking in some loop then the loop should also contain Sleep(...) statements that free processing resources for the system. Otherwise the script will unnecessarily consume processing time while waiting for the think process to end. If you don't intend to perform any other script operations during waiting for the think process to end then better use | |

| | |
|--|-------------------------|
| | SleepExec(...) instead. |
|--|-------------------------|

| | |
|------------|--|
| Command | void ExitMemBrain() |
| Purpose | Exit MemBrain |
| Parameters | None |
| Comments | MemBrain will immediately quit when this command is executed. Unsaved data will be lost without request. Further commands in the script will be ignored. |

| | |
|------------|---------------------------------|
| Command | void MinimizeWin() |
| Purpose | Minimize MemBrain's main window |
| Parameters | None |
| Comments | None |

| | |
|------------|---------------------------------|
| Command | void MaximizeWin() |
| Purpose | Maximize MemBrain's main window |
| Parameters | None |
| Comments | None |

| | |
|------------|---|
| Command | void RestoreWin() |
| Purpose | Set MemBrain's main window to the last user defined size. |
| Parameters | None |
| Comments | None |

| | |
|------------|--|
| Command | void MinimizeLessonEditor() |
| Purpose | Minimize MemBrain's Lesson Editor window |
| Parameters | None |
| Comments | None |

| | |
|------------|---|
| Command | void RestoreLessonEditor() |
| Purpose | Restore Set MemBrain's LessonEditor window. |
| Parameters | None |
| Comments | None |

| | |
|---------|-----------------------------|
| Command | void SuspendScript() |
|---------|-----------------------------|

| | |
|------------|---|
| | |
| Purpose | Suspends script execution until the user resumes it |
| Parameters | None |
| Comments | None |

| | |
|------------|-------------------------------------|
| Command | void AbortScript() |
| Purpose | Aborts script execution immediately |
| Parameters | None |
| Comments | None |

| | |
|--------------|---|
| Function | void EvaluateNetError() |
| Purpose | Re-calculate the net error of the currently loaded net based on the current Net Error Lesson. |
| Parameters | none |
| Return value | none |
| Comments | The current net error can be obtained in the script at any time using the function GetNetError(). |

| | |
|--------------|---|
| Function | double GetNetError() |
| Purpose | Return the last known net error. |
| Parameters | none |
| Return value | The net error. < 0 if the net error is unknown |
| Comments | You can use this function to supervise or record the net error during teaching for example. Another possibility is to check the net error after execution of the command ThinkLesson(). |

| | |
|--------------|--|
| Function | double GetTrainError() |
| Purpose | Return the last known training error. |
| Parameters | none |
| Return value | The training error. < 0 if the training error is unknown |
| Comments | You can use this function to supervise or record the training error during teaching for example. If the net error lesson is chosen to be the same lesson as the training lesson then the training error evaluates to the same as the net error. |

| | |
|------------|--|
| Command | void ShowErrorViewer(bool show) |
| Purpose | Shows or hides the Net Error Viewer |
| Parameters | show true shows the net error viewer, false hides it. |

| | |
|----------|------|
| Comments | None |
|----------|------|

| | | |
|------------|--------------------------------|--|
| Command | void ResetErrorViewer() | |
| Purpose | Resets the Error Viewer | |
| Parameters | None | |
| Comments | None | |

| | | |
|------------|---|---|
| Command | void NetErrorViewerLogScale(bool log) | |
| Purpose | Toggles the Net Error Viewer Y scale between logarithmic and linear display | |
| Parameters | log | true enables logarithmic scaling, false switches to linear scaling. |
| Comments | None | |

| | | |
|------------|---|--|
| Command | void ShowPatternErrorViewer(bool show) | |
| Purpose | Shows or hides the Pattern Error Viewer | |
| Parameters | show | true shows the pattern error viewer, false hides it. |
| Comments | None | |

| | | |
|--------------|---|---|
| Command | bool SelectPatternErrorViewerData(uint lessonOutColumnNum) | |
| Purpose | Select the lesson output column (and thus also the corresponding output neuron of the net) that shall be monitored in the Pattern Error Viewer. | |
| Parameters | lessonOutColumnNum | Number of the output column of the Net Error Lesson to be displayed in the Pattern Error Viewer. Must always be > 0. The most left output column in the Lesson Editor has the number 1. |
| Return value | true if successful | |
| Comments | If you are not sure about the number of a certain output column you can request its name using the function GetLessonOutputName(...). | |

| | | |
|------------|---|--|
| Command | bool SetPatternErrorViewerRange(uint fromPattern, uint toPattern) | |
| Purpose | Set the data range with respect to pattern numbers being displayed in the Pattern Error Viewer. | |
| Parameters | fromPattern | First pattern to be displayed in the graph |
| | toPattern | Last pattern to be displayed in the graph |
| Return | true if successful | |

| | |
|----------|------|
| value | |
| Comments | none |

| | |
|------------|---|
| Command | void PatternErrorViewerFitYScale() |
| Purpose | Execute autoscaling for Y-axis of pattern error viewer |
| Parameters | None |
| Comments | Call this function whenever the Pattern Error Viewer shall perform an auto scale adjustment of its Y-axis |

| | | |
|--------------|---|--|
| Command | bool SelectPatternErrorViewerLesson(EPatternViewerLesson lesson) | |
| Purpose | Select the lesson to be displayed in the Pattern Error Viewer as either the training lesson or the net error lesson | |
| Parameters | lesson | May be one of the following enumeration constants: <ul style="list-style-type: none"> • TRAIN_LESSON • NET_ERR_LESSON |
| Return value | true if successful, false on error | |
| Comments | If the parameter is set to 'NET_ERR_LESSON' but the net error lesson in the Lesson Editor is set to the same lesson as used for training then the function will still return true. However, the Pattern Error Viewer will indicate selection of the training lesson via its radio buttons. Since training and net error lesson are identical in this case, this is tolerated. | |

| | |
|--------------|---|
| Function | int Now() |
| Purpose | Get the current date and time in units of seconds since 01 Jan 1970. |
| Parameters | none |
| Return value | Number of seconds elapsed since 01 Jan 1970 |
| Comments | This function can be used to build own timers with seconds resolution or to output date/time information in the trace window or in text files. Note: This function returns a value that can be converted to a human readable date/time string using the function TimeToString(...) |

| | |
|------------|---|
| Function | void TimeToString(int time, string &out timeStr) |
| Purpose | Convert a time value into a human readable string. The time value is expected to be in units of seconds since 01 Jan 1970. |
| Parameters | time - The time value to be converted into a string timeStr - Reference to a string object that will receive the date/time string. |
| Comments | The function will create a string in the format according to the |

| | |
|--|---|
| | following example. 01 May 2009, 14:37:11 |
|--|---|

| | |
|--------------|--|
| Function | bool SetCsvFileSeparators(string &in listSep, string &in decSep) |
| Purpose | Set the separator characters used for CSV file parsing |
| Parameters | listSep - The separator used for separating columns in the CSV decSep - The separator which identifies a decimal point in the CSV |
| Return value | true if successful, false on error. The same criteria apply as when setting the separators through the GUI |
| Comments | These settings will be permanently applied, i.e. saved when MemBrain exits! It is recommended to read the corresponding settings into variables and restore them when the script exits. |

| | |
|------------|--|
| Function | void GetCsvFileSeparators(string &out listSep, string &out decSep) |
| Purpose | Get the separator characters used for CSV file parsing |
| Parameters | listSep - The separator used for separating columns in the CSV decSep - The separator which identifies a decimal point in the CSV |
| Comments | none |

| | |
|--------------|--|
| Method | void RemoveDefaultFilePwd() |
| Purpose | Removed the currently set default password from MemBrain |
| Parameters | None |
| Return value | None |
| Comments | See corresponding menu function |

| | |
|--------------|---|
| Method | bool SetDefaultPwd(const string& in pwd) |
| Purpose | Set the default password for files in MemBrain |
| Parameters | pwd The password to be used for all file load and save functions in MemBrain |
| Return value | true if successful |
| Comments | See corresponding menu function The length of the provided password must be between 1 and 32 characters. |

| | |
|--------|--|
| Method | bool EnableScriptedElements(const string& in scrElemFile) |
|--------|--|

| | |
|--------------|--|
| Purpose | Enables scripted elements using the given script file |
| Parameters | scrElemFile Script file implementing the scripted elements functions |
| Return value | true if successful |
| Comments | See corresponding menu function |

| | |
|--------------|---|
| Method | void DisableScriptedElements() |
| Purpose | Disables scripted elements |
| Parameters | None |
| Return value | None |
| Comments | See corresponding menu function |

| | |
|--------------|--|
| Method | bool EnableMultithreading(bool enableMT, enableLessonBasedMT) |
| Purpose | Enable or disable multi threading (for thinking/teaching) |
| Parameters | enableMT If true then multi threading is enabled in general. If false then multi threading is completely disabled. enableLessonBasedMT If true then lesson based multi threading is enabled. If false then lesson based multi threading is disabled. This parameter cannot be true in case the parameter <enableMT> is false. |
| Return value | true if successful. |
| Comments | See corresponding menu function |

Handling Neural Nets

All commands related directly to the whole neural net are described in this section.

| | |
|------------|---|
| Command | void NewFile() |
| Purpose | Create a new, empty net document (*.mbn) file. |
| Parameters | None |
| Comments | An unsaved net will be lost without notice when this command is executed! |

| | |
|------------|--|
| Command | void OpenNet(const string &in fileName) |
| Purpose | Open a MemBrain net (*.mbn) file |
| Parameters | fileName |
| Comments | An unsaved net will be lost without notice if a new net is opened! |

| | |
|------------|---|
| Command | void SaveNet(const string &in fileName) |
| Purpose | Save the current MemBrain net to file (*.mbn) |
| Parameters | fileName (optional) |
| Comments | If the file name is omitted then the net is stored with its current file name, i.e. the one it was opened with or the one it was last saved with. |

| | |
|--------------|---|
| Command | bool IsNetModified() |
| Purpose | Check if the currently open net has been modified (is unsaved) |
| Parameters | None |
| Return value | true if the net has not been saved after the last modification |
| Comments | This command can for example be used before to issue an 'OpenNet' command in case you want to warn the user that changes in an unsaved net will be lost due to loading another net from file. |

| | |
|--------------|---|
| Command | void GetCurrentNetFileName(string& out fileName) |
| Purpose | Get the file name of the currently loaded net. |
| Parameters | fileName string variable receiving the file name |
| Return value | none |
| Comments | none |

| | |
|------------|--|
| Command | void ExportNet(const string &in fileName) |
| Purpose | Export the current MemBrain net to csv file |
| Parameters | fileName |
| Comments | The set of exported net features is not adjustable through script. The current net export configuration is used as adjusted manually within MemBrain during the last net export. |

| | |
|------------|---|
| Command | void RandomizeNet() |
| Purpose | Randomize the current MemBrain net. |
| Parameters | None |

| | |
|----------|-------|
| Comments | None. |
|----------|-------|

| | |
|------------|---|
| Command | SetRandomizeMethodWeights(ERandomizeMethod method, double range, double offset) |
| Purpose | Configure the randomization function for the current net with respect to weights of links |
| Parameters | <p>method The method to use for randomization. Can be one of the following:</p> <ul style="list-style-type: none"> • RM_RANDOM • RM_GAUSS <p>range The range (+/- around the offset) to use for randomization Must be >= 0. For the method RM_RANDOM this is the absolute range to select random values from. For RM_GAUSS this is the Sigma to be used for the Gauss distribution to select from.</p> <p>offset The offset (center value) to be used for the random distribution to select from (typically, 0 is used as offset)</p> |
| Comments | <p>Returns true on success, i.e. if the parameters are within the given valid range.</p> <p>The configuration is stored together with the currently open net (as part of the net specific settings).</p> |

| | |
|------------|---|
| Command | SetRandomizeMethodThresholds(ERandomizeMethod method, double range, double offset) |
| Purpose | Configure the randomization function for the current net with respect to thresholds of neurons |
| Parameters | <p>method The method to use for randomization. Can be one of the following:</p> <ul style="list-style-type: none"> • RM_RANDOM • RM_GAUSS <p>range The range (+/- around the offset) to use for randomization Must be >= 0. For the method RM_RANDOM this is the absolute range to select random values from. For RM_GAUSS this is the Sigma to be used for the Gauss distribution to select from.</p> <p>offset The offset (center value) to be used for the random distribution to select from (typically, 0 is used as offset)</p> |
| Comments | <p>Returns true on success, i.e. if the parameters are within the given valid range.</p> <p>The configuration is stored together with the currently open net (as part of the net specific settings).</p> |

| | |
|---------|--|
| Command | void Shotgun() |
| Purpose | Use the Shotgun feature on the current MemBrain net. |

| | |
|------------|-------------|
| Parameters | None |
| Comments | None. |

| | |
|------------|---|
| Command | bool SelectShotgun(EShotgunDistribution distribution, EShotgunType type) |
| Purpose | Select the Shotgun distribution and type. |
| Parameters | <p>distribution The shotgun distribution to use. Can be one of the following:</p> <ul style="list-style-type: none"> • SHOTGUN_DISTRIBUTE_HOMOGENEOUS • SHOTGUN_DISTRIBUTE_GAUSS <p>type The type of shotgun to be used. Can be one of the following:</p> <ul style="list-style-type: none"> • SHOTGUN_PERCENTAGED • SHOTGUN_ABSOLUTE |
| Comments | Returns true on success, i.e. if the parameters are within the given valid range. |

| | |
|------------|--|
| Command | bool ConfigureShotgunHomogeneous(EShotgunType type, double maxPerShot) |
| Purpose | Configure the homogeneous Shotgun feature. |
| Parameters | <p>type The type of homogeneous shotgun that shall be configured. Can be one of the following:</p> <ul style="list-style-type: none"> • SHOTGUN_PERCENTAGED • SHOTGUN_ABSOLUTE <p>maxPerShot The maximum value for each weight or threshold change. For the absolute type this value must be between 0 and 1000. For the percentaged type the value must be between 0 and 10000.</p> |
| Comments | Returns true on success, i.e. if the parameters are within the given valid range. |

| | |
|------------|---|
| Command | bool ConfigureShotgunGaussian(EShotgunType type, double stdDev, double centerValue) |
| Purpose | Configure the gaussian Shotgun feature. |
| Parameters | <p>type The type of gaussian shotgun that shall be configured. Can be one of the following:</p> <ul style="list-style-type: none"> • SHOTGUN_PERCENTAGED • SHOTGUN_ABSOLUTE <p>stdDev The Standard Deviation for the gaussian characteristic. For the absolute type this value must be between 0 and 1000. For the percentaged type the value must be between 0 and 10000.</p> <p>centerValue The Center Value for the gaussian characteristic. For the absolute type this value must be between 0 and 1000. For the percentaged</p> |

| | |
|----------|---|
| | type the value must be between 0 and 10000. |
| Comments | Returns true on success, i.e. if the parameters are within the given valid range. |

| | |
|------------|---|
| Command | void ResetNet() |
| Purpose | Reset the current MemBrain net (set all neuron activations and all residual activation spikes on all links to 0). |
| Parameters | None |
| Comments | None. |

| | |
|--------------|--|
| Command | uint GetInputCount() |
| Purpose | Get the number of input neurons in the net |
| Parameters | None |
| Return value | Number of input neurons in the net |
| Comments | None. |

| | |
|--------------|---|
| Command | uint GetOutputCount() |
| Purpose | Get the number of output neurons in the net |
| Parameters | None |
| Return value | Number of output neurons in the net |
| Comments | None. |

| | | |
|--------------|---|--|
| Command | bool GetInputName(uint inNeuronNum, string &out name) | |
| Purpose | Get the name of an input neuron | |
| Parameters | inNeuronNum | Number of the input neuron. Must always be > 0. The top left input neuron has the number 1. Other input neurons have increasing numbers assigned in lines from top to bottom and from left to right. |
| | name | string variable receiving the name if successful |
| Return value | true if successful | |
| Comments | If you are not sure about the number of a certain input neuron you can request its name using the function GetInputName(...). | |

| | | |
|------------|--|--|
| Command | bool GetOutputName(uint outNeuronNum, string &out name) | |
| Purpose | Get the name of an output neuron | |
| Parameters | outNeuronNum | Number of the output neuron. Must always be > 0. The top left output neuron has the number 1. Other output |

| | | |
|--------------|---|--|
| | | neurons have increasing numbers assigned in lines from top to bottom and from left to right. |
| | name | string variable receiving the name if successful |
| Return value | true if successful | |
| Comments | If you are not sure about the number of a certain output neuron you can request its name using the function <code>GetOutputName(...)</code> . | |

| | | |
|--------------|---|---|
| Command | bool ApplyInputAct(uint inNeuronNum, double activation) | |
| Purpose | Apply an activation value to an input neuron | |
| Parameters | inNeuronNum | Number of the input neuron. Must always be > 0. The top left input neuron has the number 1. Other input neurons have increasing numbers assigned in lines from top to bottom and from left to right. |
| | activation | floating point value applied to the activation of the input neuron. The valid activation range for a neuron is always defined by its activation function (if no normalization is used for the neuron) or its user defined normalization range (if normalization is activated for the neuron). If the provided value exceeds the valid limits then the neuron's activation will be clipped to the corresponding limit. |
| Return value | true if successful | |
| Comments | <p>If you are not sure about the number of a certain input neuron you can request its name using the function <code>GetInputName(...)</code>.</p> <p>Use function <code>GetInputActRange(...)</code> if you are not sure about the valid activation range for an input neuron</p> | |

| | | |
|--------------|---|--|
| Command | bool GetInputAct(uint inNeuronNum, double &out activation) | |
| Purpose | Get the current activation of an input neuron | |
| Parameters | inNeuronNum | Number of the input neuron. Must always be > 0. The top left input neuron has the number 1. Other input neurons have increasing numbers assigned in lines from top to bottom and from left to right. |
| | activation | floating point variable receiving the activation if successful |
| Return value | true if successful | |
| Comments | If you are not sure about the number of a certain input neuron you can request its name using the function <code>GetInputName(...)</code> . | |

| | | |
|------------|---|---|
| Command | bool GetOutputAct(uint outNeuronNum, double &out activation) | |
| Purpose | Get the current activation of an output neuron | |
| Parameters | outNeuronNum | Number of the output neuron. Must always be > 0. The top left output neuron has the number 1. Other output neurons have increasing numbers assigned in lines from top to bottom and from left to right. |

| | | |
|--------------|---|--|
| | activation | floating point variable receiving the activation if successful |
| Return value | true if successful | |
| Comments | If you are not sure about the number of a certain output neuron you can request its name using the function <code>GetOutputName(...)</code> . | |

| | | |
|--------------|---|---|
| Command | bool GetOutputOut(uint outNeuronNum, double &out output) | |
| Purpose | Get the last known output signal value of an output neuron | |
| Parameters | outNeuronNum | Number of the output neuron. Must always be > 0. The top left output neuron has the number 1. Other output neurons have increasing numbers assigned in lines from top to bottom and from left to right. |
| | output | floating point variable receiving the output value if successful |
| Return value | true if successful | |
| Comments | If you are not sure about the number of a certain output neuron you can request its name using the function <code>GetOutputName(...)</code> . | |

| | | |
|--------------|---|--|
| Command | bool GetInputActRange(uint inNeuronNum, double &out minAct, double &out maxAct) | |
| Purpose | Get the valid activation range of an input neuron | |
| Parameters | inNeuronNum | Number of the input neuron. Must always be > 0. The top left input neuron has the number 1. Other input neurons have increasing numbers assigned in lines from top to bottom and from left to right. |
| | minAct | floating point variable receiving the minimum valid activation value if successful |
| | maxAct | floating point variable receiving the maximum valid activation value if successful |
| Return value | true if successful | |
| Comments | If you are not sure about the number of a certain input neuron you can request its name using the function <code>GetInputName(...)</code> . | |

| | | |
|--------------|---|---|
| Command | bool GetOutputActRange(uint inNeuronNum, double &out minAct, double &out maxAct) | |
| Purpose | Get the valid activation range of an output neuron | |
| Parameters | inNeuronNum | Number of the output neuron. Must always be > 0. The top left output neuron has the number 1. Other output neurons have increasing numbers assigned in lines from top to bottom and from left to right. |
| | minAct | floating point variable receiving the minimum valid activation value if successful |
| | maxAct | floating point variable receiving the maximum valid activation value if successful |
| Return value | true if successful | |
| Comments | If you are not sure about the number of a certain output neuron you | |

| | |
|--|---|
| | can request its name using the function <code>GetOutputName(...)</code> . |
|--|---|

| | |
|--------------|---|
| Method | void RemoveNetFilePwd() |
| Purpose | Disables password protection for the currently open neural net. |
| Parameters | None |
| Return value | None |
| Comments | Disable password protection before to save a net to disk if you want the file to be stored without password protection. You only need this command in case you already had a password assigned to the net beforehand. |

| | |
|--------------|---|
| Method | bool SetNetFilePwd(const string& in pwd) |
| Purpose | Set the password to be used for loading or saving the current net. |
| Parameters | pwd The password to be used for the current net for all Load and Save operations. |
| Return value | true if successful |
| Comments | When a net is loaded from file then the provided password only is of effect in case the loaded file has been detected to be encrypted. Else the password is automatically removed from the net after loading the net from file. The length of the provided password must be between 1 and 32 characters. |

Handling Group Relations

All commands related to group relations and associated sub nets are described in this section.

| | |
|--------------|---|
| Command | bool ChooseSubNet(const string& in relationName) |
| Purpose | Choose the currently active relation/sub net via its name |
| Parameters | relationName Name of the relation and associated sub net to choose as the active one or "" for setting the whole net to be active |
| Return value | true if successful |
| Comments | This script function relates to the manual function of choosing the currently active relation/sub net . |

Editing Neural Nets

This section covers all commands that are used to edit a net

| | |
|--------------|---|
| Command | uint GetHiddenLayerCount() |
| Purpose | Get the number of normal hidden layers in the net |
| Parameters | None |
| Return value | Number of normal hidden layers in the net |
| Comments | <p>Note that this function only returns the number of NORMAL hidden layers. There might be hidden neurons in a net that are not assigned to the normal hidden layers. I.e. there might be context neurons in a net. These neurons would be in the Context Layer instead of the the normal hidden layers.</p> <p>Also, as long as the output of a hidden neuron is not connected it will be located in the UNRESOLVED layer and not in a normal hidden layer.</p> <p>In general, the following rules apply to location of hidden neurons:</p> <ol style="list-style-type: none"> 1.) Output not connected: Neuron is located in the UNRESOLVED layer 2.) Input not connected but output connected: Neuron is located in the CONTEXT layer 3.) Input and output connected. Input link is not a loopback link: Neuron is located in a NORMAL HIDDEN layer |

| | |
|--------------|---|
| Command | uint GetHiddenCount(uint layerNum) |
| Purpose | Get the number of neurons in the given normal hidden layer |
| Parameters | layerNum Number of the normal hidden layer. Must be 1 .. GetHiddenLayerCount() |
| Return value | Number of neurons in the hidden layer |
| Comments | <p>Note that this function only returns the number of neurons in a given NORMAL hidden layer. There might be hidden neurons in a net that are not assigned to the normal hidden layers.</p> <p>See command GetHiddenLayerCount() for more details on hidden neuron layer assignment</p> |

| | |
|--------------|--|
| Command | uint GetHiddenCount() |
| Purpose | Get the number of neurons in all normal hidden layers |
| Parameters | None |
| Return value | Number of neurons in all normal hidden layers |
| Comments | <p>Note that this function only returns the number of neurons in the NORMAL hidden layers. There might be hidden neurons in a net that are not assigned to the normal hidden layers.</p> <p>See command GetHiddenLayerCount() for more details on hidden neuron layer assignment</p> |

| | |
|--------------|--|
| Command | uint GetContextCount() |
| Purpose | Get the number of neurons in the context layer of the net |
| Parameters | None |
| Return value | Number of neurons in the context layer |
| Comments | See command GetHiddenLayerCount() for more details on hidden neuron layer assignment |

| | |
|--------------|--|
| Command | uint GetUnresolvedCount() |
| Purpose | Get the number of neurons in the unresolved layer of the net |
| Parameters | None |
| Return value | Number of neurons in the context layer |
| Comments | See command GetHiddenLayerCount() for more details on hidden neuron layer assignment |

| | |
|--------------|--|
| Command | void ClearSelection() |
| Purpose | Clear the current selection (so that nothing is selected anymore, neither neurons nor links) |
| Parameters | None |
| Return value | None |
| Comments | None |

| | |
|--------------|---|
| Command | uint SelectNeuronsByName(const string& in neuronName, bool addToSelection, bool findMultiple) |
| Purpose | Select neuron(s) by their name |
| Parameters | neuronName - The name of the neuron(s) to select addToSelection - If <true> then the neurons are selected additionally to the current selection. <false> clears the current selection first. findMultiple - Optional parameter. If <true> then all neurons with the given name are selected. If <false> or if parameter is omitted then the search stops at the first found neuron with this name. |
| Return value | number of found (and thus selected) neurons. |
| Comments | <ul style="list-style-type: none"> - If the neuron(s) with the specified name are already selected the function still returns the number of found neurons. - If <findMultiple> is false the function always returns either 0 or 1. - If some neurons are already selected and <addToSelection> is <true> then the return value does not necessarily reflect the final number of selected neurons in the net. It always represents the neurons found. |

| | |
|--------------|---|
| Command | bool SelectInput(uint neuronNum, bool addToSelection) |
| Purpose | Select an input neuron |
| Parameters | neuronNum - The number of the input neuron. Must be 1 .. GetInputCount() addToSelection - If <true> then the neuron is selected additionally to the current selection. <false> clears the current selection first. |
| Return value | <true> if successful, <false> if specified neuron does not exist (parameter out of range). |
| Comments | None |

| | |
|--------------|--|
| Command | void SelectInput(bool addToSelection) |
| Purpose | Select all input neurons |
| Parameters | addToSelection - If <true> then the neurons are selected additionally to the current selection. <false> clears the current selection first. |
| Return value | None |
| Comments | None |

| | |
|--------------|---|
| Command | bool SelectOutput(uint neuronNum, bool addToSelection) |
| Purpose | Select an output neuron |
| Parameters | neuronNum - The number of the output neuron. Must be 1 .. GetOutputCount() addToSelection - If <true> then the neuron is selected additionally to the current selection. <false> clears the current selection first. |
| Return value | <true> if successful, <false> if specified neuron does not exist (parameter out of range). |
| Comments | None |

| | |
|--------------|--|
| Command | void SelectOutput(bool addToSelection) |
| Purpose | Select all output neurons |
| Parameters | addToSelection - If <true> then the neurons are selected additionally to the current selection. <false> clears the current selection first. |
| Return value | None |
| Comments | None |

| | |
|---------|--|
| Command | bool SelectHidden(uint layerNum, uint neuronNum, bool |
|---------|--|

| | |
|--------------|--|
| | addToSelection) |
| Purpose | Select a neuron in a normal hidden layer |
| Parameters | <p>layerNum - The number of the normal hidden layer - Must be 1 .. GetHiddenLayerCount()</p> <p>neuronNum - The number of the hidden neuron. Must be 1 .. GetHiddenCount(layerNum)</p> <p>addToSelection - If <true> then the neuron is selected additionally to the current selection. <false> clears the current selection first.</p> |
| Return value | <true> if successful, <false> if specified neuron does not exist (parameter out of range). |
| Comments | See command GetHiddenLayerCount() for more details on hidden neuron layer assignment |

| | |
|--------------|---|
| Command | bool SelectHidden(uint layerNum, bool addToSelection) |
| Purpose | Select all neurons in a normal hidden layer |
| Parameters | <p>layerNum - The number of the normal hidden layer - Must be 1 .. GetHiddenLayerCount()</p> <p>addToSelection - If <true> then the neurons are selected additionally to the current selection. <false> clears the current selection first.</p> |
| Return value | <true> if successful, <false> if specified normal hidden layer does not exist (parameter out of range). |
| Comments | See command GetHiddenLayerCount() for more details on hidden neuron layer assignment |

| | |
|--------------|--|
| Command | void SelectHidden(bool addToSelection) |
| Purpose | Select all neurons in all normal hidden layers |
| Parameters | addToSelection - If <true> then the neurons are selected additionally to the current selection. <false> clears the current selection first. |
| Return value | None |
| Comments | See command GetHiddenLayerCount() for more details on hidden neuron layer assignment |

| | |
|------------|--|
| Command | bool SelectContext(uint neuronNum, bool addToSelection) |
| Purpose | Select a context neuron |
| Parameters | <p>neuronNum - The number of the context neuron. Must be 1 .. GetContextCount()</p> <p>addToSelection - If <true> then the neuron is selected additionally to the current selection. <false> clears the current selection first.</p> |

| | |
|--------------|--|
| Return value | <true> if successful, <false> if specified neuron does not exist (parameter out of range). |
| Comments | None |

| | |
|--------------|--|
| Command | void SelectContext(bool addToSelection) |
| Purpose | Select all context neurons |
| Parameters | addToSelection - If <true> then the neurons are selected additionally to the current selection. <false> clears the current selection first. |
| Return value | None |
| Comments | None |

| | |
|--------------|---|
| Command | bool SelectUnresolved(uint neuronNum, bool addToSelection) |
| Purpose | Select an unresolved neuron |
| Parameters | neuronNum - The number of the unresolved neuron. Must be 1 .. GetUnresolvedCount() addToSelection - If <true> then the neuron is selected additionally to the current selection. <false> clears the current selection first. |
| Return value | <true> if successful, <false> if specified neuron does not exist (parameter out of range). |
| Comments | None |

| | |
|--------------|--|
| Command | void SelectUnresolved(bool addToSelection) |
| Purpose | Select all unresolved neurons |
| Parameters | addToSelection - If <true> then the neurons are selected additionally to the current selection. <false> clears the current selection first. |
| Return value | None |
| Comments | None |

| | |
|--------------|--|
| Command | void Select(uint16 x, uint16 y, bool addToSelection) |
| Purpose | Select an object according to the given point on the screen |
| Parameters | x - X-Position of the point y - Y-Position of the point addToSelection - If <true> then the object is selected additionally to the current selection. <false> clears the current selection first. |
| Return value | None |
| Comments | None |

| | |
|--------------|---|
| Command | void Select(uint16 l, uint16 t, uint16 r, uint16 b, bool addToSelection) |
| Purpose | Select objects according to the given rectangle on the screen |
| Parameters | l - Left coordinate of the rectangle t - Top coordinate of the rectangle r - Right coordinate of the rectangle b -Bottom coordinate of the rectangle addToSelection - If <true> then the objects are selected additionally to the current selection. <false> clears the current selection first. |
| Return value | None |
| Comments | None |

| | |
|--------------|---|
| Command | void ExtraSelection() |
| Purpose | Apply Extra Selection to all neurons that are currently selected. |
| Parameters | None |
| Return value | None |
| Comments | None |

| | |
|--------------|---|
| Command | void ClearExtraSelection() |
| Purpose | Clear the Extra Selection (so that no neurons are Extra Selected anymore) |
| Parameters | None |
| Return value | None |
| Comments | None |

| | |
|--------------|---|
| Command | void AddInput(uint posX, uint posY, string &in name) |
| Purpose | Add an input neuron to the net |
| Parameters | posX - The X Position of the center of the new neuron. Must be 0 .. 30000 posY - The Y Position of the center of the new neuron. Must be 0 .. 30000 name - The name of the new neuron. Specify "" (empty string) to cause the neuron to keep the creation default name (which is derived from the internal neuron ID). |
| Return value | None |
| Comments | If the option ' Snap to Grid ' in MemBrain's view menu is active then |

| | |
|--|---|
| | <p>the neuron will be positioned on the grid point nearest to the specified position.</p> <ul style="list-style-type: none"> - New input neurons will always be created with their corresponding default properties. See data structure SNeuronProp for a description of the default parameters |
|--|---|

| | |
|--------------|--|
| Command | void AddOutput(uint posX, uint posY, string &in name) |
| Purpose | Add an output neuron to the net |
| Parameters | <p>posX - The X Position of the center of the new neuron. Must be 0 .. 30000</p> <p>posY - The Y Position of the center of the new neuron. Must be 0 .. 30000</p> <p>name - The name of the new neuron. Specify "" (empty string) to cause the neuron to keep the creation default name (which is derived from the internal neuron ID).</p> |
| Return value | None |
| Comments | <ul style="list-style-type: none"> - If the option 'Snap to Grid' in MemBrain's view menu is active then the neuron will be positioned on the grid point nearest to the specified position. - New output neurons will always be created with their corresponding default properties. See data structure SNeuronProp for a description of the default parameters |

| | |
|--------------|---|
| Command | void AddHidden(uint posX, uint posY, string &in name) |
| Purpose | Add a hidden (unresolved) neuron to the net |
| Parameters | <p>posX - The X Position of the center of the new neuron. Must be 0 .. 30000</p> <p>posY - The Y Position of the center of the new neuron. Must be 0 .. 30000</p> <p>name - The name of the new neuron. Specify "" (empty string) to cause the neuron to keep the creation default name (which is derived from the internal neuron ID).</p> |
| Return value | None |
| Comments | <ul style="list-style-type: none"> - If the option 'Snap to Grid' in MemBrain's view menu is active then the neuron will be positioned on the grid point nearest to the specified position. - New hidden neurons will always be created with their corresponding default properties. See data structure SNeuronProp for a description of the default parameters - New hidden neurons will always be placed in the unresolved layer since their output has not yet been connected! See command |

| | |
|--|--|
| | GetHiddenLayerCount() for more details on hidden neuron layer assignment |
|--|--|

| | |
|--------------|--|
| Command | bool GetSelectedNeuronProp(SNeuronProp &out prop) |
| Purpose | Get the properties of the currently selected neuron |
| Parameters | prop - The SNeuronProp data structure to read the neuron parameters into. See description of the data structure further down in this chapter for details on the available struct members. |
| Return value | <true> on success, <false> on error (i.e. not exactly one neuron selected). |
| Comments | There must be exactly one neuron selected when the command is executed. Else the function will not read any properties and return <false>. |

| | |
|--------------|--|
| Command | void SetSelectedNeuronProp(SNeuronProp &in prop) |
| Purpose | Set the properties of all currently selected neurons |
| Parameters | prop - The SNeuronProp data structure to take the neuron parameters from. See description of the data structure further down in this chapter for details on the available struct members. |
| Return value | None |
| Comments | <p>- Note that there may be multiple neurons selected. All selected neurons will be assigned the new parameters.</p> <p>- When the struct contains invalid parameter values these are automatically clipped to valid ranges.</p> <p>- The recommended approach to use this function is to read the neuron properties of one neuron into a data structure, then modify only the desired subset of all properties by modifying the structure and use the SetSelectedNeuronProp command to write the modified data to a set of neurons.</p> <p>Example:</p> <pre>// Select the first output neuron SelectOutput(1, false); SNeuronProp prop; // Create a new neuron // properties structure // Read the properties of the first output neuron. // Then modify the struct to use a different // activation function, select all output neurons // and write the modified properties struct back // to all of them. if (GetSelectedNeuronProp(prop)) { prop.actFunc = AF_TAN_H; // Modify this member of the struct SelectOutput(false); // Select all }</pre> |

| | |
|--|--|
| | <pre> output neurons SetSelectedNeuronProp(prop); // Write modified properties to all selected neurons } else { Trace("Unable to read neuron properties!\n"); AbortScript(); } </pre> |
|--|--|

| | |
|--------------|---|
| Command | bool GetSelectedNeuronPos(int16 &out x, int16 &out y) |
| Purpose | Get the position of the currently selected neuron |
| Parameters | x - The X-Position of the center of the selected neuron y - The Y-Position of the center of the selected neuron |
| Return value | <true> on success, <false> on error (i.e. not exactly one neuron selected). |
| Comments | There must be exactly one neuron selected when the command is executed. Else the function will not read the position and return <false>. x and y can be between 0 and 30000. |

| | |
|--------------|--|
| Command | void SetSelectedNeuronName(string &in name) |
| Purpose | Set the name of all currently selected neurons |
| Parameters | name - The name that shall be assigned to all selected neurons. |
| Return value | None |
| Comments | None |

| | |
|--------------|--|
| Command | void MoveSelectedNeurons(int16 deltaX, int16 deltaY) |
| Purpose | Move all selected neurons by the specified vector |
| Parameters | deltaX - Distance to move neurons in X direction deltaY - Distance to move neurons in Y direction |
| Return value | None |
| Comments | Positive values for deltaX move to the right. Positive values for deltaY move downwards. |

| | |
|------------|--|
| Command | bool GetSelectedLinkProp(SLinkProp &out prop) |
| Purpose | Get the properties of the currently selected link |
| Parameters | prop - The SLinkProp data structure to read the link parameters into. See description of the data structure further down in this chapter for details on the available struct members. |

| | |
|--------------|--|
| Return value | <true> on success, <false> on error (i.e. not exactly one link selected). |
| Comments | There must be exactly one link selected when the command is executed. Else the function will not read any properties and return <false>. |

| | |
|--------------|--|
| Command | void SetSelectedLinkProp(SLinkProp &in prop) |
| Purpose | Set the properties of all currently selected links |
| Parameters | prop - The SLinkProp data structure to take the link parameters from. See description of the data structure further down in this chapter for details on the available struct members. |
| Return value | None |
| Comments | <p>- Note that there may be multiple links selected. All selected links will be assigned the new parameters.</p> <p>- When the struct contains invalid parameter values these are automatically clipped to valid ranges.</p> |

| SNeuronProp | | | | | |
|---|-----------|---|---------------------------|------------------------|----------------------------|
| - Neuron Properties Data Structure and Default Values - | | | | | |
| Property (struct member) Name | Data Type | Valid Range | Default for Input Neurons | Default Output Neurons | Default for Hidden Neurons |
| act | double | - | 0 | 0 | 0 |
| actFunc | EActFunc | AF_LOGISTIC AF_IDENTICAL AF_IDENTICAL_0_1 AF_TAN_H AF_BINARY AF_MIN_EUC LID_DIST AF_IDENTICAL_M11 AF_RELU AF_SOFTPLUS AF_BIN_DIFF AF_SOFTMAX AF_SOFTSIGN AF_ELU AF_GELU | AF_IDENTICAL | AF_LOGISTIC | AF_LOGISTIC |

| | | | | | |
|----------------------------------|------------------|---|---------|---------|---------|
| inputFunc | EInputFunc | IF_SUM IF_MUL IF_MAX IF_AVG IF_RAND | IF_SUM | IF_SUM | IF_SUM |
| actThres | double | - | 0 | 0 | 0 |
| lockActThres | bool | - | true | false | false |
| actSustain | double | 0 .. 1 | 1.0 | 0 | 0 |
| outputFireLevel | EOutputFireLevel | OFL_1 OFL_ACT | OFL_ACT | OFL_ACT | OFL_ACT |
| outputRecovTime | uint | 1 .. 100000 | 1 | 1 | 1 |
| fireThresLow | double | fireThresHi must be >= fireThresLo | -1.0 | -1.0 | -1.0 |
| fireThresHi | double | | -1.0 | -1.0 | -1.0 |
| useNormalization | bool | - | false | false | false |
| normRangeLow | double | - | -1.0 | 0 | 0 |
| normRangeHigh | double | - | 1.0 | 1.0 | 1.0 |
| useActIgnoreVal | bool | - | false | false | false |
| actIgnoreVal | double | - | 2.0 | 2.0 | 2.0 |
| expLogistic | double | - | 3.0 | 3.0 | 3.0 |
| parmTanHyp | double | - | 3.0 | 3.0 | 3.0 |
| leakage | double | >= 0 | 0.01 | 0.01 | 0.01 |
| binDiffSlope | double | >= 0 | 100 | 100 | 100 |
| alphaElu | double | >= 0 | 1.0 | 1.0 | 1.0 |
| allowTeacherOutputConnect | bool | - | true | true | true |
| displayName | bool | - | true | true | false |
| displayAct | bool | - | true | true | false |
| isPixel | bool | - | false | false | false |
| width | uint16 | 15 - 30000 | 45 | 45 | 45 |

| | |
|--------------|--|
| Command | void SelectLinksFromExtra() |
| Purpose | Select all links that route From the Extra Selection to the Selection . |
| Parameters | None |
| Return value | None |
| Comments | Like with the corresponding command in MemBrain all other selected objects are selected. |

| | |
|---------|--|
| Command | void SelectLinksToExtra() |
| Purpose | Select all links that route from the Selection To the Extra Selection . |

| | |
|--------------|---|
| Parameters | None |
| Return value | None |
| Comments | Like with the corresponding command in MemBrain all other selected objects are deleted. |

| | |
|--------------|--|
| Command | void ConnectFromExtra() |
| Purpose | Connect all neurons From the Extra Selection to the Selection . I.e. the outputs of all Extra Selected neurons are connected to all inputs of all Selected neurons. |
| Parameters | None |
| Return value | None |
| Comments | Note that Selection and Extra Selection may overlap: A neuron can be Extra Selected and Selected at the same time! |

| | |
|--------------|---|
| Command | void ConnectToExtra() |
| Purpose | Connect all neurons from the Selection To the Extra Selection . I.e. the outputs of all Selected neurons are connected to all inputs of all Extra Selected neurons. |
| Parameters | None |
| Return value | None |
| Comments | Note that Selection and Extra Selection may overlap: A neuron can be Extra Selected and Selected at the same time! |

| SLinkProp | | | |
|---|-----------|-------------|---------------|
| - Link Properties Data Structure and Default Values - | | | |
| Property (struct member) Name | Data Type | Valid Range | Default Value |
| <i>weight</i> | double | - | 0.5 |
| <i>lockWeight</i> | bool | - | false |
| <i>length</i> | uint16 | 1 .. 10000 | 1 |
| <i>displayWeight</i> | bool | - | false |

| | |
|------------|--|
| Command | void DeleteSelectedObjects() |
| Purpose | Delete all selected objects from the net |
| Parameters | None |
| Return | None |

| | |
|----------|------|
| value | |
| Comments | None |

| | |
|--------------|--|
| Command | uint16 GetGridWidth() |
| Purpose | Get the currently adjusted grid width of MemBrain |
| Parameters | None |
| Return value | The grid width that can be adjusted in MemBrain using the menu command <View><Set Grid Width...> |
| Comments | None |

| | |
|--------------|---|
| Command | bool SetGridWidth(uint16 width) |
| Purpose | Set/Adjust the grid width of MemBrain |
| Parameters | width - The new grid width to be used (valid range: 5 - 500) |
| Return value | true on success |
| Comments | Same function as MemBrain's menu command <View><Set Grid Width...> |

| | |
|--------------|--|
| Command | void GetGridPoint(uint16 x, uint16 y, uint16 &out gridX, uint16 &out gridY) |
| Purpose | Get the nearest grid point to a given point |
| Parameters | x - X-Coordinate of the input point y - Y-Coordinate of the input point gridX - X-Coordinate of the nearest grid point gridY - Y-Coordinate of the nearest grid point |
| Return value | None |
| Comments | When the resulting point would be > 30000 in any of the coordinates these will be clipped to 30000 no matter if this coordinate value lies on a grid point or not. |

| NeuronArrayDefinition | | | | |
|--|-----------|-------------|---------|---|
| - Definition for a neuron array and default values - | | | | |
| Property (class member) Name | Data Type | Valid Range | Default | Comment |
| dimX | uint | 1 - 100000 | 5 | Number of neurons in x (horizontal) dimension |
| dimY | uint | 1 - 100000 | 5 | Number of neurons in y (vertical) |

| | | | | |
|--------------------------------|-------------|------------------------------------|---|--|
| | | | | dimension |
| distX | uint | 0 - 10000 | 60 | Distance of neurons (center to center) in x dimension |
| distY | uint | 0 - 10000 | 60 | Distance of neurons (center to center) in y dimension |
| convolutionalThresholds | bool | false true | false | Set this to true if the activation thresholds of the neurons shall be created as convolutions (i.e. one common activation threshold) |
| neuronTemplate | SNeuronProp | See members of SNeuronProp | As currently adjusted in MemBrain via <Edit><Default properties...> | The template neuron used to create the new neurons in the array/matrix |
| neuronType | ENeuronType | NT_INPUT NT_HIDDEN NT_OUTPUT | | |

| | | |
|--------------|---|--|
| Command | bool AddNeuronArray(const NeuronArrayDefinition@ def) | |
| Purpose | Add an array (or matrix) of neurons to the net | |
| Parameters | def - The definition of the neuron array / matrix - See NeuronArrayDefinition for details | |
| Return value | true on success | |
| Comments | Same function as MemBrain's menu command <Insert><Neuron Matrix...> | |

| | | |
|--------------|--|------------------------|
| Command | bool AddDelayNeurons(uint linkLenMin, uint linkLenMax) | Parameter range |
| Purpose | Add a set of delay neurons to each of the currently selected neurons | |
| Parameters | linkLenMin - The minimum link len for the new delay neurons | 2 .. 10000 |
| | linkLenMax - The maximum link len for the new delay neurons | 2 .. 10000 |
| Return value | true on success | |
| Comments | Same function as MemBrain's menu command <Insert><Delay Neurons...> For each selected neuron and for each link len step a new delay | |

| | |
|--|----------------------|
| | neuron is generated. |
|--|----------------------|

| Command | bool AddDecayNeurons(double sustainFactMin, double sustainFactMax, uint decayNeuronCount) | Parameter range |
|--------------|---|-------------------|
| Purpose | Add a set of delay neurons to each of the currently selected neurons | |
| Parameters | sustainFactMin - The minimum sustain factor for the new decay neurons | 0.00001 - 0.99999 |
| | sustainFactMax - The maximum sustain factor for the new decay neurons | 0.00001 - 0.99999 |
| | decayNeuronCount - The number of decay neurons to add to each of the selected source neurons | 1 - 100000 |
| Return value | true on success | |
| Comments | Same function as MemBrain's menu command <Insert><Decay Neurons...> | |

| Command | bool AddIntegratorNeurons(uint capacityMin, uint capacityMax, uint integratorNeuronCount) | Parameter range |
|--------------|---|-----------------|
| Purpose | Add a set of integrator neurons to each of the currently selected neurons | |
| Parameters | capacityMin - The minimum Think Step capacity for the new integrator neurons | > 1 |
| | capacityMax - The maximum Think Step capacity for the new integrator neurons | > 1 |
| | integratorNeuronCount - The number of integrator neurons to add to each of the selected source neurons | 1 - 100000 |
| Return value | true on success | |
| Comments | Same function as MemBrain's menu command <Insert><Integrator Neurons...> | |

| Command | bool AddNeuronLayer(const NeuronLayerDefinition@ def, uint maxRandOutLinksPerNeuron, const MatrixConnectSpec@ connectSpec) |
|------------|---|
| Purpose | Add a new neuron layer, i.e. a new neuron matrix including specific links routing from the currently selected neurons to the new neuron layer. |
| Parameters | def - The definition of the new neuron layer - See NeuronLayerDefinition for details maxRandOutLinksPerNeuron - Maximum number of outputs links per selected neuron. Used with connectMode MCM_RANDOM only. Else value is don't care. - See NeuronLayerDefinition for details connectSpec - The matrix connection spec used with connectMode MCM_GROUPED only. Else values don't care. |

| | |
|--------------|--|
| | |
| Return value | true on success |
| Comments | Same function as MemBrain's menu command <Insert><Neuron Layer...> |

| NeuronLayerDefinition | | | | |
|--|--------------------------|---|-------------|---|
| - Definition for a neuron layer and default values - | | | | |
| Property (class member) Name | Data Type | Valid Range | Default | Comment |
| arrayDef | NeuronArrayDefinition | - | - | The array which describes the number and properties of the neurons in the new layer See documentation on NeuronArrayDefinition |
| linkPropTemplate | SLinkProp | - | - | Template used to create the new links leading to the new neuron layer See documentation of SLinkProp |
| createMode | ENeuronLayerCreationMode | NLCM_MANUAL creates a new layer with the layout given by the member <i>arrayDef</i> NLCM_AUTO creates a new layer with automatic number and layout based on the currently selected neurons. For connection modes MCM_FULL, MCM_1_1 and MCM_RAND the layout will be identical to | NLCM_MANUAL | The data <i>dimX</i> and <i>dimY</i> in <i>arrayDef</i> are ignored for NLCM_AUTO. NLCM_AUTO only works in case the currently selected neurons for a rectangular matrix. |

| | | | | |
|--------------------|---------------------|---|----------|---|
| | | <p>the layout of the currently selected neurons.</p> <p>For connection mode MCM_GROUPED the layout will be calculated so that the connection scheme defined by the parameter <code>connectSpec</code> of the command <code>AddNeuronLayer</code> can be established.</p> | | |
| connectMode | EMatrixConnect Mode | <p>MCM_FULL Full interconnection of all selected neurons to all neurons in the new layer.</p> <p>MCM_1_1 1:1 connection of all selected neurons to the neurons in the new layer. Requires <code>createMode NLCM_AUTO</code></p> <p>MCM_RAND Random connection of all selected neurons to the neurons in the new layer.</p> <p>MCM_GROUPED Grouped connection of all selected neurons to the neurons in the new layer.</p> | MCM_FULL | See documentation of command <code>AddNeuronLayer</code> for more details |

| | |
|--------------|--|
| Command | void CopyToClipboard() |
| Purpose | Copy the currently selected content to the clipboard |
| Parameters | None |
| Return value | N/A |
| Comments | Same as <Copy> function in MemBrain (Ctrl + 'C') |

| | |
|--------------|--|
| Command | void PasteClipboard(bool leaveAttachedToCursor = false) |
| Purpose | Paste the content of the clipboard |
| Parameters | leaveAttachedToCursor Optional parameter, is false when not provided. If set to true then the pasted content stays attached to the mouse (i.e. same behavior as when using the function via the user interface of MemBrain. |
| Return value | N/A |
| Comments | Same as <Paste> function in MemBrain (Ctrl + 'V') |

HandlingLessonData

This section and its sub sections cover all available commands to handle Lesson data through MemBrain scripts.

The commands in the tables below refer to control of the Lesson Editor in MemBrain. I.e., they act exactly as when the user would interact with the Lesson Editor directly. This is a good approach for issuing a limited number of lesson related processing commands at a time. **For more performant lesson data creation and management** the script class '[Lesson](#)' should be used instead which is described [here](#). This script class allows to handle lesson data 'behind the scenes' without the need for time consuming screen updates in the lesson editor.

Note:

Many of the commands described in the following tables cause the Lesson Editor to show up when executed. If you want to be sure that the Lesson Editor is hidden after some lesson related commands then use the command ShowLessonEditor(false); afterwards.

| | | |
|------------|---|--|
| Command | void ShowLessonEditor(bool show) | |
| Purpose | Shows/Hides the Lesson Editor | |
| Parameters | show | true = show Lesson Editor, false = hide the Lesson Editor. |
| Comments | None | |

| | |
|---------|---|
| Command | void EnableLessonEditorUpdate(bool enable) |
|---------|---|

| | |
|------------|--|
| | |
| Purpose | Enables/disables automatic update of the Lesson Editor display |
| Parameters | enable true = enable, false = disable Lesson Editor update. |
| Comments | <p>Call this function with parameter enable = false if you perform a large number of changes with respect to the Lesson Editor data. This will increase performance significantly. After all changes have been performed call the function again, this time with parameter enable = true to re-enable the update of the Lesson Editor on the screen. Calling the function with parameter enable = true will immediately trigger an update of the Lesson Editor display.</p> <p>After script execution finishes or when the script execution is suspended then the Lesson Editor update will get enabled again if it has been disabled.</p> |

| | |
|------------|---|
| Command | void SetLessonCount(uint count) |
| Purpose | Sets the number of Lessons administered by the Lesson Editor |
| Parameters | count (must be > 0) |
| Comments | <p>The Lesson Editor always has one Lesson open at a minimum. You only need this command if you want to use more than one lesson in your script, e.g. for recording network output to a separate lesson while using the input of another lesson.</p> <p>If you reduce the number of administered lessons all lessons at a position higher than the new count are deleted automatically!</p> |

| | |
|--------------|---|
| Command | uint GetLessonCount() |
| Purpose | Returns the number of Lessons administered by the Lesson Editor |
| Parameters | none |
| Return value | Current number of lessons in the Lesson Editor |
| Comments | <p>The Lesson Editor always has one Lesson open at a minimum. You only need this command if you want to use more than one lesson in your script, e.g. for recording network output to a separate lesson while using the input of another lesson.</p> <p>If you reduce the number of administered lessons all lessons at a position higher than the new count are deleted automatically!</p> |

| | |
|------------|---|
| Command | void SelectLesson(uint number) |
| Purpose | Selects the currently active lesson in the Lesson Editor. |
| Parameters | number (must be > 0 and <= number of administered lessons, see command GetLessonCount()) |
| Comments | <p>Use this command to set one of the lessons administered by the Lesson Editor as the active one. The active lesson also is the one used for teaching,</p> <p>There are several commands that always act on the currently active lesson.</p> |

| | |
|--------------|--|
| Command | uint GetSelectedLesson() |
| Purpose | Returns the number of the currently active lesson in the Lesson Editor. |
| Parameters | none |
| Return value | Number of currently selected (active) lesson in the Lesson Editor |
| Comments | Use this command to set one of the lessons administered by the Lesson Editor as the active one. The active lesson also is the one used for teaching, There are several commands that always act on the currently active lesson. |

| | |
|------------|---|
| Command | void SelectNetErrLesson(uint number) |
| Purpose | Sets the lesson in the Lesson Editor used to calculate the net error . |
| Parameters | number (must be ≥ 0 and \leq number of administered lessons, see command LESSON_COUNT) A value of '0' adjusts the net error lesson to be the same as the currently active lesson (default setting after MemBrain is started). A value > 0 sets the net error lesson to the specified number. From then on the net error lesson is fixed to this given number until changed again to another number or until set back to be the active lesson through a parameter value of '0'. |
| Comments | Use this command to set one of the lessons administered by the Lesson Editor as the lesson for calculation of the net error during training. There are several commands that always act on the currently active lesson. |

| | |
|--------------|--|
| Command | uint GetSelectedNetErrLesson() |
| Purpose | Returns the number of the lesson in the Lesson Editor which is used to calculate the net error. |
| Parameters | none |
| Return value | Number of lesson in the lesson editor that is used to calculate the net error |
| Comments | Use this command to set one of the lessons administered by the Lesson Editor as the lesson for calculation of the net error during training. There are several commands that always act on the currently active lesson. |

| | |
|------------|---|
| Command | void EnableLessonOutData(bool enable) |
| Purpose | Enables/disables the output data section of the currently active lesson in the Lesson Editor. |
| Parameters | enable true = enable outputs, false = disable outputs |

| | |
|----------|--|
| Comments | See here for more details on this functionality. |
|----------|--|

| | |
|------------|--|
| Command | void LoadLesson(const string &in fileName) |
| Purpose | Loads the currently active lesson from a MemBrain Lesson File (*.mbl) |
| Parameters | fileName |
| Comments | The existing content in the active lesson in the Lesson Editor will be overwritten/lost! |

| | |
|------------|---|
| Command | void LoadLesson(const Lesson@ source) |
| Purpose | Loads the currently active lesson from a MemBrain Lesson object |
| Parameters | source Reference to a Lesson object the currently active lesson in the Lesson Editor shall be loaded from |
| Comments | The existing content in the active lesson in the Lesson Editor will be overwritten/lost! |

| | |
|------------|---|
| Command | bool LoadLesson(const Lesson@ source, uint targetNum) |
| Purpose | Loads the lesson with the given number in the Lesson Editor from a MemBrain Lesson object |
| Parameters | source Reference to a Lesson object the target lesson in the Lesson Editor shall be loaded from targetNum Number of the target lesson in the Lesson Editor |
| Return | true if successful |
| Comments | The existing content in the target lesson in the Lesson Editor will be overwritten/lost! |

| | |
|------------|--|
| Command | void AppendLesson(const string &in fileName) |
| Purpose | Appends the data in the given lesson file (*.mbl) to the currently active lesson |
| Parameters | fileName |
| Comments | None |

| | |
|------------|--|
| Command | void SaveLesson(const string &in fileName) |
| Purpose | Saves the currently active lesson to a MemBrain Lesson File (*.mbl) |
| Parameters | fileName (optional) |
| Comments | If the file name is omitted then the lesson is stored with its current |

| | |
|--|---|
| | file name, i.e. the one it was opened with or the one it was last saved with. If no file name is available the command will fail. |
|--|---|

| | |
|------------|---|
| Command | void ImportLesson(const string &in fileName) |
| Purpose | Imports the currently active lesson from a MemBrain Sectioned Lesson CSV File |
| Parameters | fileName |
| Comments | None |

| | |
|------------|---|
| Command | void ExportLesson(const string &in fileName, uint maxCols) |
| Purpose | Exports the currently active lesson to a MemBrain Sectioned Lesson CSV File |
| Parameters | 1. fileName 2. maxColumnCount (optional) |
| Comments | The number of columns in the export file can be limited by specifying a second, optional parameter. This is useful because some spreadsheet programs limit the available number of columns to e.g. 256. |

| | |
|------------|---|
| Command | void ImportLessonRaw(const string &in fileName) |
| Purpose | Imports the currently active lesson from a MemBrain Raw Lesson CSV File |
| Parameters | fileName |
| Comments | The number of inputs and outputs in the Lesson Editor has to be adjusted to the number of columns read from the csv file, otherwise the import will fail. |

| | |
|------------|---|
| Command | void ExportLessonRaw(const string &in fileName, uint maxColumnCount) |
| Purpose | Exports the currently active lesson to a MemBrain Raw Lesson CSV File |
| Parameters | 1. fileName 2. maxColumnCount (optional) |
| Comments | The number of columns in the export file can be limited by specifying a second, optional parameter. This is useful because some spreadsheet programs limit the available number of columns to e.g. 256. |

| | |
|------------|---|
| Command | void ImportLessonInputsRaw(const string &in fileName) |
| Purpose | Imports the input data of the currently active lesson from a MemBrain Raw Lesson CSV File |
| Parameters | fileName |

| | |
|----------|---|
| Comments | The number of inputs in the Lesson Editor has to be adjusted to the number of columns read from the csv file, otherwise the import will fail. |
|----------|---|

| | |
|------------|---|
| Command | void ExportLessonInputsRaw(const string &in fileName, uint maxColumnCount) |
| Purpose | Exports the input data of the currently active lesson to a MemBrain Raw Lesson CSV File |
| Parameters | 1. fileName 2. maxColumnCount (optional) |
| Comments | The number of columns in the export file can be limited by specifying a second, optional parameter. This is useful because some spreadsheet programs limit the available number of columns to e.g. 256. |

| | |
|------------|--|
| Command | void ImportLessonOutputsRaw(const string &in fileName) |
| Purpose | Imports the output data of the currently active lesson from a MemBrain Raw Lesson CSV File |
| Parameters | fileName |
| Comments | The number of outputs in the Lesson Editor has to be adjusted to the number of columns read from the csv file, otherwise the import will fail. |

| | |
|------------|---|
| Command | void ExportLessonOutputsRaw(const string &in fileName, uint maxColumnCount) |
| Purpose | Exports the output data of the currently active lesson to a MemBrain Raw Lesson CSV File |
| Parameters | 1. fileName 2. maxColumnCount (optional) |
| Comments | The number of columns in the export file can be limited by specifying a second, optional parameter. This is useful because some spreadsheet programs limit the available number of columns to e.g. 256. |

| | |
|------------|--|
| Command | void SetLessonInputCount(uint count) |
| Purpose | Sets the number of input neurons for the currently active lesson in the Lesson Editor. |
| Parameters | count (must be ≥ 0) |
| Comments | Use this command to set the number of input neurons of the currently active Lesson in the Lesson Editor. If you want to import a Raw Lesson CSV file it is required that the number of input and output neurons matches the number of neuron names contained in the Raw Lesson CSV file for example. |

| | |
|------------|---|
| Command | uint GetLessonInputCount() |
| Purpose | Returns the number of input neurons for the currently active lesson in the Lesson Editor. |
| Parameters | none |
| Comments | none |

| | |
|------------|---|
| Command | void SetLessonOutputCount(uint count) |
| Purpose | Sets the number of output neurons for the currently active lesson in the Lesson Editor. |
| Parameters | count (must be ≥ 0) |
| Comments | Use this command to set the number of output neurons of the currently active Lesson in the Lesson Editor. If you want to import a Raw Lesson CSV file it is required that the number of input and output neurons matches the number of neuron names contained in the Raw Lesson CSV file for example. |

| | |
|------------|--|
| Command | uint GetLessonOutputCount() |
| Purpose | Returns the number of output neurons for the currently active lesson in the Lesson Editor. |
| Parameters | none |
| Comments | none |

| | |
|------------|---|
| Command | void NamesFromNet() |
| Purpose | Takes the names of the input and output neurons of the loaded net and assigns them to the currently active lesson in the Lesson Editor. |
| Parameters | None |
| Comments | Same functionality as button <Names From Net> on the Lesson Editor |

| | |
|--------------|---|
| Command | bool NamesFromNet(Lesson@ target) |
| Purpose | Takes the names of the input and output neurons of the loaded net and assigns them to the given script lesson object |
| Parameters | target Reference to the script lesson object which will be formatted according to the currently loaded net. |
| Return value | true if successful |
| Comments | None |

| | |
|---------|--------------------------|
| Command | void NamesToNet() |
|---------|--------------------------|

| | |
|------------|---|
| Purpose | Takes the names of the currently active lesson in the Lesson Editor and assigns them to the input and output neurons of the loaded net. |
| Parameters | None |
| Comments | Same functionality as button <Names To Net> on the Lesson Editor |

| | |
|--------------|---|
| Command | bool NamesToNet(Lesson@ source) |
| Purpose | Takes the names of the given script lesson lesson and assigns them to the input and output neurons of the currently loaded net. |
| Parameters | source Reference to script lesson object to take the I/O names from |
| Return value | true if successful |
| Comments | Same functionality as button <Names To Net> on the Lesson Editor |

| | |
|------------|---|
| Command | void ClearLesson() |
| Purpose | Deletes all patterns of the currently active lesson in the Lesson Editor. |
| Parameters | None |
| Comments | None |

| | |
|------------|---|
| Command | void SelectPattern(uint number) |
| Purpose | Selects the currently active pattern of the currently active lesson in the Lesson Editor. |
| Parameters | number (must be > 0 and <= number of available patterns in the active lesson) |
| Comments | Use this command to set one of the patterns in the active lesson as the active one. There are several commands that always act on the currently active pattern of the active lesson. |

| | |
|--------------|---|
| Command | uint GetSelectedPattern() |
| Purpose | Returns the number of the selected pattern of the active lesson in the Lesson Editor. |
| Parameters | none |
| Return value | number of the selected pattern in the currently active lesson |
| Comments | none |

| | |
|---------|---|
| Command | void DeletePattern() |
| Purpose | Deletes the active pattern of the currently active lesson in the Lesson Editor. |

| | |
|------------|-------------|
| Parameters | None |
| Comments | None |

| | |
|------------|--|
| Command | void ApplyPattern() |
| Purpose | Applies the active pattern of the currently active lesson in the Lesson Editor to the input neurons of the net. |
| Parameters | None |
| Comments | Note that this command does not perform a think step it just assigns activation values to the net's input Neurons. |

| | | |
|--------------|--|--|
| Command | void PatternFromNet(bool addPattern) | |
| Purpose | Reads the current activations from the input and output neurons of the net and assigns them to the corresponding input and output columns of the active lesson in the Lesson Editor. | |
| Parameters | addPattern | if <true> then a new pattern is appended to the end of the lesson. If <false> then the currently active pattern in the Lesson Editor is overwritten with the data read from the net. |
| Return value | true if successful | |
| Comments | See here for more details on this functionality. | |

| | | |
|------------|--|--|
| Command | void SetRecordingType(ERecType recordingType) | |
| Purpose | Specifies what to record to a lesson with respect to the output neurons of the net: It can be selected whether the activation values or the output signals shall be recorded. | |
| Parameters | recordingType | RT_ACT - Activations values are recorded or RT_OUT - Output signals are recorded |
| Comments | See here for more details on this functionality. | |

| | | |
|------------|--|--|
| Command | void StartRecording(uint lessonNum, uint stepCount) | |
| Purpose | Enables recording of data to a lesson | |
| Parameters | <p>1. LessonNumber The number of the lesson to add recorded data to. (must be > 0 and <= number of administered lessons, see function GetLessonCount())</p> <p>2. StepCount (optional) The number of think steps after which a new pattern shall be recorded. If not specified then 1 is used which records a new pattern after every think step of the net.</p> | |
| Comments | This command can be used to evaluate the reaction of the net to input data from another lesson: Specify recording to an empty lesson, set | |

| | |
|--|--|
| | the lesson with the input data as the active one and use the function ThinkLesson(). Then export the recorded lesson to csv for example. |
|--|--|

| | |
|------------|--|
| Command | void StopRecording() |
| Purpose | Disables recording of data to a lesson |
| Parameters | None |
| Comments | None |

| | |
|--------------|--|
| Command | uint GetLessonSize() |
| Purpose | Returns the size (number of patterns) in the currently active lesson in the Lesson Editor. |
| Parameters | none |
| Return value | number of patterns in the currently active lesson |
| Comments | none |

| | |
|------------|--|
| Command | void SetLessonSize(uint size) |
| Purpose | Set the size (number of patterns) of the currently active lesson in the Lesson Editor. |
| Parameters | size The new size (number of patterns) for the lesson |
| Comments | Existing data is deleted in case the new size is smaller than the current size of the active lesson! |

| | |
|--------------|--|
| Command | uint SetLessonSize(uint size) |
| Purpose | Sets the number of patterns in the currently active lesson in the Lesson Editor. |
| Parameters | size New size for the lesson |
| Return value | none |
| Comments | If the new size is smaller than the current size the currently selected pattern is changed to the last pattern in the now smaller lesson |

| | |
|--------------|--|
| Command | uint GetNetErrLessonSize() |
| Purpose | Returns the number of patterns in the currently selected Net Error lesson. |
| Parameters | none |
| Return value | number of patterns in the currently selected Net Error lesson |
| Comments | none |

| | | |
|--------------|--|---|
| Command | void SetLessonInputName(uint inColumnNum, const string &in name) | |
| Purpose | Set the name of an input column in the currently active lesson in the Lesson Editor. | |
| Parameters | inColumnNum | Number of the input column. Must always be > 0. The most left input column in the Lesson Editor has the number 1. |
| | name | Name to be assigned to the input column |
| Return value | none | |
| Comments | There is no return value to this function since MemBrain will abort the script automatically when an error occurs that has effect on the loaded data (see explanations here). | |

| | | |
|--------------|--|---|
| Command | bool GetLessonInputName(uint inColumnNum, string &out name) | |
| Purpose | Get the name of an input column in the currently active lesson in the Lesson Editor. | |
| Parameters | inColumnNum | Number of the input column. Must always be > 0. The most left input column in the Lesson Editor has the number 1. |
| | name | string variable receiving the name if successful |
| Return value | true if successful | |
| Comments | none | |

| | | |
|--------------|--|---|
| Command | void SetLessonOutputName(uint outColumnNum, const string &in name) | |
| Purpose | Set the name of an output column in the currently active lesson in the Lesson Editor. | |
| Parameters | outColumnNum | Number of the output column. Must always be > 0. The most left output column in the Lesson Editor has the number 1. |
| | name | Name to be assigned to the output column |
| Return value | none | |
| Comments | There is no return value to this function since MemBrain will abort the script automatically when an error occurs that has effect on the loaded data (see explanations here). | |

| | | |
|--------------|---|---|
| Command | bool GetLessonOutputName(uint outColumnNum, string &out name) | |
| Purpose | Get the name of an output column in the currently active lesson in the Lesson Editor. | |
| Parameters | outColumnNum | Number of the output column. Must always be > 0. The most left output column in the Lesson Editor has the number 1. |
| | name | string variable receiving the name if successful |
| Return value | true if successful | |
| Comments | none | |

| | | |
|--------------|--|---|
| Command | void SetPatternInput(uint inColumnNum, double value) | |
| Purpose | Set the value of an input column in the currently active pattern of the currently active lesson in the Lesson Editor. | |
| Parameters | inColumnNum | Number of the input column. Must always be > 0. The most left input column in the Lesson Editor has the number 1. |
| | value | The floating point value to be assigned to the column of the active pattern |
| Return value | none | |
| Comments | There is no return value to this function since MemBrain will abort the script automatically when an error occurs that has effect on the loaded data (see explanations here). | |

| | | |
|--------------|---|---|
| Command | bool GetPatternInput(uint inColumnNum, double &out value) | |
| Purpose | Get the value of an input column in the currently active pattern of the currently active lesson in the Lesson Editor. | |
| Parameters | inColumnNum | Number of the input column. Must always be > 0. The most left input column in the Lesson Editor has the number 1. |
| | value | The floating point variable that will receive the pattern value if successful |
| Return value | true if successful | |
| Comments | none | |

| | | |
|--------------|--|---|
| Command | void SetPatternOutput(uint outColumnNum, double value) | |
| Purpose | Set the value of an output column in the currently active pattern of the currently active lesson in the Lesson Editor. | |
| Parameters | outColumnNum | Number of the output column. Must always be > 0. The most left output column in the Lesson Editor has the number 1. |
| | value | The floating point value to be assigned to the column of the active pattern |
| Return value | none | |
| Comments | There is no return value to this function since MemBrain will abort the script automatically when an error occurs that has effect on the loaded data (see explanations here). | |

| | | |
|------------|--|---|
| Command | bool GetPatternOutput(uint outColumnNum, double &out value) | |
| Purpose | Get the value of an output column in the currently active pattern of the currently active lesson in the Lesson Editor. | |
| Parameters | outColumnNum | Number of the output column. Must always be > 0. The most left output column in the Lesson Editor has the |

| | | |
|--------------|--------------------|---|
| | | number 1. |
| | value | The floating point variable that will receive the pattern value if successful |
| Return value | true if successful | |
| Comments | none | |

| | | |
|------------|---|--|
| Command | void AddPattern(uint count) | |
| Purpose | Add new patterns to the end of the current lesson. | |
| Parameters | Number of patterns to add (optional) | |
| Comments | All values in the new patterns will be set to zeroes. If called without the parameter <count> then one pattern is added (same as if count = 1 was used). | |

| | | |
|--------------|--|---|
| Command | bool GetLessonInputMinMax(uint columnNum, double &out min, double &out max) | |
| Purpose | Get the minimum and the maximum value of a given input column in the active lesson. | |
| Parameters | columnNum | Number of the input column. Must always be > 0. The most left input column in the Lesson Editor has the number 1. |
| | min | The floating point variable that will receive the minimum pattern value |
| | max | The floating point variable that will receive the maximum pattern value |
| Return value | true if successful | |
| Comments | none | |

| | | |
|--------------|---|---|
| Command | bool GetLessonOutputMinMax(uint columnNum, double &out min, double &out max) | |
| Purpose | Get the minimum and the maximum value of a given input column in the active lesson. | |
| Parameters | columnNum | Number of the output column. Must always be > 0. The most left output column in the Lesson Editor has the number 1. |
| | min | The floating point variable that will receive the minimum pattern value |
| | max | The floating point variable that will receive the maximum pattern value |
| Return value | true if successful | |
| Comments | none | |

| | | |
|---------|---|--|
| Command | bool SplitLesson(double percent, bool byRandom = true) | |
|---------|---|--|

| | | |
|--------------|--|--|
| Purpose | Split off a percentage portion of the currently selected lesson into a newly created lesson. Use random selection for the split action | |
| Parameters | percent | The percentage portion of the current lesson that will be split off and moved to the new lesson. |
| | byRandom | Optional parameter: If true (default value) then the patterns which are split off are selected by random. Else a percentage tail of the lesson is split off. Use this option for splitting off parts of lessons for time variant nets: The order of the patterns is important here and must be kept. |
| Return value | true if successful, i.e. if at least one pattern has been split off. false if no pattern has been split off. | |
| Comments | Works the same as the menu item <Extras><Split Current Lesson...> from the Lesson Editor 's menu. If successful then a new lesson with the split off patterns will be created and added to the Lesson Editor. I.e. the number of lessons in the Lesson Editor will increase by one. | |

| | | |
|--------------|---|--|
| Command | bool CreateInputAverageLesson(uint newInputCount) | |
| Purpose | Create a new lesson with a reduced number of inputs by using averaging of the inputs of the currently active lesson. | |
| Parameters | newInputCount | The number of inputs that shall be present in the new lesson |
| Return value | true is successful, false on error. | |
| Comments | For more details on the behaviour of these functions see section 'Averaging Inputs' in the Lesson Editor. | |

| | | |
|------------|--|-----------------------------|
| Command | void SetLessonName(const string &in name) | |
| Purpose | Set the name of the active lesson | |
| Parameters | name | The new name for the lesson |
| Comments | Note that this is not the file name but the given name of the lesson shown and editable in the lesson editor | |

| | | |
|------------|--|--------------------------------|
| Command | void SetLessonComment(const string &in comment) | |
| Purpose | Set the comment of the active lesson | |
| Parameters | comment | The new comment for the lesson |
| Comments | None | |

| | | |
|---------|---|--|
| Command | bool SetPatternName(const string &in name) | |
| Purpose | Set the name of the active pattern in the active lesson | |

| | | |
|--------------|--|------------------------------|
| Parameters | name | The new name for the pattern |
| Return value | true if successful | |
| Comments | In order for the function to succeed the current lesson must not be empty, i.e. at least one pattern must be present | |

| | | |
|--------------|--|---------------------------------|
| Command | bool SetPatternComment(const string &in comment) | |
| Purpose | Set the comment of the active pattern in the active lesson | |
| Parameters | comment | The new comment for the pattern |
| Return value | true if successful | |
| Comments | In order for the function to succeed the current lesson must not be empty, i.e. at least one pattern must be present | |

| | | |
|------------|--|--|
| Command | void GetLessonName(string &out name) | |
| Purpose | Get the name of the active lesson | |
| Parameters | name | The string variable that will receive the name of the lesson |
| Comments | Note that this is not the file name but the given name of the lesson shown and editable in the lesson editor | |

| | | |
|------------|---|---|
| Command | void GetLessonComment(string &out comment) | |
| Purpose | Get the comment of the active lesson | |
| Parameters | comment | The string variable that will receive the comment of the lesson |
| Comments | None name | |

| | | |
|--------------|--|---|
| Command | bool GetPatternName(string &out name) | |
| Purpose | Get the name of the active pattern in the active lesson | |
| Parameters | name | The string variable that will receive the name of the pattern |
| Return value | true if successful | |
| Comments | In order for the function to succeed the current lesson must not be empty, i.e. at least one pattern must be present | |

| | | |
|--------------|---|--|
| Command | bool GetPatternComment(string &out comment) | |
| Purpose | Get the comment of the active pattern in the active lesson | |
| Parameters | comment | The string variable that will receive the comment of the pattern |
| Return value | true if successful | |
| Comments | In order for the function to succeed the current lesson must not be | |

| | |
|--|--|
| | empty, i.e. at least one pattern must be present |
|--|--|

| | | |
|--------------|--|--|
| Command | bool CopyPatterns(uint sourceNum, uint targetNum, uint sourceStartNum, uint targetStartNum, uint count) | |
| Purpose | Copy a specified number of patterns from a source lesson to a target lesson in the lesson editor | |
| Parameters | sourceNum | Number of the source lesson according to Lesson Editor |
| | targetNum | Number of the source lesson according to Lesson Editor |
| | sourceStartNum | Pattern number were to start copying in the source lesson |
| | targetStartNum | Pattern number were to start pasting to in the target lesson |
| | count | Number of patterns to copy |
| Return value | true if successful | |
| Comments | The number of input and output columns must be set to the same values for source and target lesson (identical format of lessons) in order to make the command to complete successfully. If the target lesson is too small to capture the copied patterns it will automatically be enlarged. | |

| | | |
|--------------|---|-------------------------------------|
| Command | bool ExportNetErrorGraph(const string &in fileName) | |
| Purpose | Export the content of the net error graph to a CSV file | |
| Parameters | fileName | The file name to export the data to |
| Return value | true if successful | |
| Comments | Same functionality as the 'Export Graph' button of the Error Graph Viewer . | |

| | | |
|--------------|---|-------------------------------------|
| Command | bool ExportPatternErrorGraph(const string &in fileName) | |
| Purpose | Export the content of the pattern error graph to a CSV file | |
| Parameters | fileName | The file name to export the data to |
| Return value | true if successful | |
| Comments | Same functionality as the 'Export Graph' button of the Pattern Error Viewer . | |

| | | |
|--------------|---|-------------------------------------|
| Command | bool ExportPatternErrorValidation(const string &in fileName) | |
| Purpose | Export the validation data used by the pattern error viewer to a CSV file | |
| Parameters | fileName | The file name to export the data to |
| Return value | true if successful | |

| | |
|----------|--|
| Comments | Same functionality as the 'Export Validation' button of the Pattern Error Viewer . |
|----------|--|

| | |
|--------------|--|
| Method | bool ReadPatternInputsFromImageFile(const string& in fileName, uint resampleCols, uint resampleRows, uint inputStartIdx, double grayDark, double grayBright) |
| Purpose | <p>Read a picture into the inputs of the current pattern of the lesson as gray scale values.</p> <p>Reading will stop if all pixels have been read or if no more lesson inputs are available.</p> <p>Before reading the image is re-sampled according to the given column and row counts.</p> |
| Parameters | <p>fileName The image file to read the picture from</p> <p>resampleCols Number of columns to apply for re-sampling. If 0 is provided then the number of columns in the original picture file are used.</p> <p>resampleRows Number of rows to apply for re-sampling. If 0 is provided then the number of rows in the original picture file are used.</p> <p>inputStartIdx (optional) 0-based index into the input array to start reading to. This will be the input position where the pixel in the upper left corner of the image is read to. Default value is 0 if parameter is not provided.</p> <p>grayDark (optional) Value used to represent dark pixels after read-in. Default value is 0 if parameter is not provided.</p> <p>grayBright (optional) Value used to represent bright pixels after read-in. Default value is 1 if parameter is not provided.</p> |
| Return value | true if successful |
| Comments | A large variety of picture file formats is supported, like bmp, jpg, gif, png, mng, ico, tif, tga, pcx, wbmp, wmf, jp2, jpc, pgx, ras, pnm |

| | |
|--------------|---|
| Method | void RemoveLessonFilePwd() |
| Purpose | Disables password protection for the current lesson in the Lesson Editor. |
| Parameters | None |
| Return value | None |
| Comments | Disable password protection before to save a lesson to disk if you want the file to be stored without password protection. You only need this command in case you already had a password assigned to the lesson object. |

| Method | bool SetLessonFilePwd(const string& in pwd) |
|--------------|--|
| Purpose | Set the password to be used for loading or saving the current lesson in the Lesson Editor. |
| Parameters | pwd The password to be used for the current lesson for all Load and Save operations. |
| Return value | true if successful |
| Comments | When a lesson is loaded from file then the provided password only is of effect in case the loaded file has been detected to be encrypted. Else the password is automatically removed from the lesson after loading the lesson from file. The length of the provided password must be between 1 and 32 characters. |

Script class for Lessons

The scripting language of MemBrain provides the class **Lesson** to create and manage MemBrain Lessons directly in the script without using the Lesson Editor remote commands. This is the preferred way of lesson data handling in the scripts in case intensive data manipulation on lessons shall be performed because it provides a much higher level of performance compared to the Lesson Editor remote control command set. Once all required lesson editing has been done the script lessons can directly be loaded into the Lesson Editor if required.

Script usage example:

```
void SomeScriptFunction()
{
    // Create the lesson object
    Lesson myLesson;

    // Copy the format and content of the currently edited lesson in the Lesson Editor into the script
    object myLesson.CloneFromLessonEditor();

    ... // Do something with the lesson and/or other lessons

    // Load the modified lesson back into the Lesson Editor
    LoadLesson(myLesson);
}
```

Important Note:

All indices are 0-based for Lesson objects in the script! This is different to the Lesson Editor related commands where 1-based numbers are used to address the loaded lessons in the Lesson Editor.

I.e. the index 0 always indicates the first element in the lesson or the first column in the input section of a lesson for instance.

Constructors

The following table lists all available ways to construct a Lesson object.

| | |
|--------------|--|
| Constructors | <p>Lesson @f() Construct an empty lesson, number of inputs = number of outputs = size = 0</p> <p>Lesson @f(uint inCount, uint outCount) Construct an empty lesson with the given number of input and output columns</p> <p>Lesson @f(uint inCount, uint outCount, uint size) Construct a lesson with the given number of input and output columns as well as the given size (number of patterns). The values in the patterns will all be set to 0 as initial value</p> |
| Comments | The size of a lesson object as well as the number of inputs and outputs can be changed after creation. For large lessons this can take some time, however. Thus, if the target size of a lesson is known upfront it is best to set it to the target value directly before the first use of the object or at creation time. This does not apply in case lessons are loaded/imported from files. In this case the lesson size is determined automatically by the framework during loading/importing from file. |

| | |
|--------------|--|
| Method | bool Lesson::Load(const string& in fileName) |
| Purpose | Load the lesson from a MemBrain Lesson File (*.mbl) |
| Parameters | fileName The lesson file to load into the Lesson object |
| Return value | true if successful |
| Comments | The existing content in the lesson will be overwritten/lost! |

| | |
|--------------|--|
| Method | bool Lesson::Save(const string& in fileName) |
| Purpose | Save the lesson to a MemBrain Lesson File (*.mbl) |
| Parameters | fileName The lesson file to save the Lesson object to |
| Return value | true if successful |
| Comments | The existing content in the lesson will be overwritten/lost! |

| | |
|--------------|---|
| Method | bool Lesson::Import(const string& in fileName) |
| Purpose | Import the lesson from a MemBrain Sectioned Lesson CSV File |
| Parameters | fileName The file to import the Lesson object from |
| Return value | true if successful |

| | |
|----------|--|
| Comments | The existing content in the lesson will be overwritten/lost! |
|----------|--|

| | |
|--------------|---|
| Method | bool Lesson::Export(const string& in fileName, uint maxColumnCount) |
| Purpose | Export the lesson to a MemBrain Sectioned Lesson CSV File |
| Parameters | fileName The file to export the Lesson object to maxColumnCount (optional) Maximum number of columns in the CSV file before to insert a line break |
| Return value | true if successful |
| Comments | The number of columns in the export file can be limited by specifying a second, optional parameter. This is useful because some spreadsheet programs limit the available number of columns to e.g. 256. |

| | |
|--------------|--|
| Method | bool Lesson::ImportRaw(const string& in fileName) |
| Purpose | Import the lesson from a MemBrain Raw Lesson CSV File |
| Parameters | fileName The file to import the Lesson object from |
| Return value | true if successful |
| Comments | The existing content in the lesson will be overwritten/lost! The number of inputs and outputs of the lesson has to be adjusted to the number of columns read from the csv file, otherwise the import will fail. |

| | |
|--------------|---|
| Method | bool Lesson::ExportRaw(const string& in fileName, uint maxColumnCount) |
| Purpose | Export the lesson to a MemBrain Raw Lesson CSV File |
| Parameters | fileName The file to export the Lesson object to maxColumnCount (optional) Maximum number of columns in the CSV file before to insert a line break |
| Return value | true if successful |
| Comments | The number of columns in the export file can be limited by specifying a second, optional parameter. This is useful because some spreadsheet programs limit the available number of columns to e.g. 256. |

| | |
|---------|--|
| Method | bool Lesson::ImportInputsRaw(const string& in fileName) |
| Purpose | Import the input data of the lesson from a MemBrain Raw Lesson CSV |

| | |
|--------------|--|
| | File |
| Parameters | fileName The file to import the Lesson object input data from |
| Return value | true if successful |
| Comments | The number of inputs of the lesson has to be adjusted to the number of columns read from the csv file, otherwise the import will fail. |

| | |
|--------------|--|
| Method | bool Lesson::ExportInputsRaw(const string& in fileName, uint maxColumnCount) |
| Purpose | Export the input data of the lesson to a MemBrain Raw Lesson CSV File |
| Parameters | fileName The file to export the Lesson object input data to maxColumnCount (optional) Maximum number of columns in the CSV file before to insert a line break |
| Return value | true if successful |
| Comments | The number of columns in the export file can be limited by specifying a second, optional parameter. This is useful because some spreadsheet programs limit the available number of columns to e.g. 256. |

| | |
|--------------|---|
| Method | bool Lesson::ImportOutputsRaw(const string& in fileName) |
| Purpose | Import the output data of the lesson from a MemBrain Raw Lesson CSV File |
| Parameters | fileName The file to import the Lesson object output data from |
| Return value | true if successful |
| Comments | The number of outputs of the lesson has to be adjusted to the number of columns read from the csv file, otherwise the import will fail. |

| | |
|--------------|---|
| Method | bool Lesson::ExportOutputsRaw(const string& in fileName, uint maxColumnCount) |
| Purpose | Export the output data of the lesson to a MemBrain Raw Lesson CSV File |
| Parameters | fileName The file to export the Lesson object output data to maxColumnCount (optional) Maximum number of columns in the CSV file before to insert a line break |
| Return value | true if successful |
| Comments | The number of columns in the export file can be limited by specifying a second, optional parameter. This is useful because some spreadsheet |

| | |
|--|---|
| | programs limit the available number of columns to e.g. 256. |
|--|---|

| | |
|--------------|---|
| Method | void Lesson::SetSize(uint size) |
| Purpose | Set the size (number of patterns) of the lesson |
| Parameters | size The new size (number of patterns) |
| Return value | None |
| Comments | If the new size is smaller than the current size then patterns in the end of the lessons will be deleted. If the new size is bigger than the current size then patterns with 0 values are added to the end of the lesson. |

| | |
|--------------|---|
| Method | uint Lesson::GetSize() |
| Purpose | Get the size (number of patterns) of the lesson |
| Parameters | None |
| Return value | The size (number of patterns) of the lesson |
| Comments | None |

| | |
|--------------|---|
| Method | bool Lesson::NamesFromNet() |
| Purpose | Synch the lesson to the currently loaded net: Automatically adjust number and names of the lesson's inputs and outputs to the I/O neurons of the net. |
| Parameters | None |
| Return value | true if successful |
| Comments | None |

| | |
|--------------|---|
| Method | void Lesson::SetInputCount(uint count) |
| Purpose | Set the number of input columns of the lesson |
| Parameters | count Number of input columns |
| Return value | None |
| Comments | If the new input columns number is smaller than the current one then input columns on the right side of the lesson are deleted. If the number of input columns is bigger than the current one then new input columns with 0-valued patterns are added to the right. |

| Method | uint Lesson::GetInputCount() |
|--------------|---|
| Purpose | Get the number of input columns of the lesson |
| Parameters | None |
| Return value | Number of input columns of the lesson |
| Comments | None |

| Method | void Lesson::SetOutputCount(uint count) |
|--------------|---|
| Purpose | Set the number of output columns of the lesson |
| Parameters | count Number of output columns |
| Return value | None |
| Comments | If the new output columns number is smaller than the current one then output columns on the right side of the lesson are deleted. If the number of output columns is bigger than the current one then new output columns with 0-valued patterns are added to the right. |

| Method | uint Lesson::GetOutputCount() |
|--------------|--|
| Purpose | Get the number of output columns of the lesson |
| Parameters | None |
| Return value | Number of output columns of the lesson |
| Comments | None |

| Method | bool Lesson::Append(const Lesson@ source) |
|--------------|--|
| Purpose | Append the given source lesson to the lesson |
| Parameters | source The lesson to append to this lesson |
| Return value | true if successful |
| Comments | The number of input and output columns of the two lessons must be identical in order to make this command work |

| Method | bool Lesson::CopyPatterns(const Lesson@ source, uint sourceStartIdx, uint targetStartIdx, uint count) |
|---------|--|
| Purpose | Copy a specified range of patterns from a source lesson to this lesson |

| | |
|-----------------|---|
| Parameter s | <p>source The lesson to copy the patterns from</p> <p>sourceStartIdx 0-based pattern index in the source lesson to start copying from</p> <p>targetStartIdx 0-based pattern index in this lesson (i.e. the target lesson) to start copying to</p> <p>count Number of patterns to copy</p> |
| Return value | true if successful |
| Comment s | <p>The number of input and output columns must be set to the same values for source and target lesson (identical format of lessons) in order to make the command to complete successfully.</p> <p>If the target lesson is too small to capture the copied patterns it will automatically be enlarged.</p> |

| | |
|-----------------|---|
| Method | void Lesson::Clone(const Lesson@ source) |
| Purpose | Clone the lesson from another lesson |
| Parameter s | <p>source The lesson to clone this lesson from</p> |
| Return value | None |
| Comment s | All data in the target lesson will be deleted/replaced |

| | |
|-----------------|--|
| Method | void Lesson::CloneFromLessonEditor() |
| Purpose | Clone the lesson from the currently edited lesson in the Lesson Editor |
| Parameter s | None |
| Return value | None |
| Comment s | All data in the target lesson will be deleted/replaced |

| | |
|-----------------|---|
| Method | bool Lesson::CloneFromLessonEditor(uint sourceNum) |
| Purpose | Clone the lesson from the given lesson number in the Lesson Editor |
| Parameter s | <p>sourceNum Number of the lesson in the Lesson Editor to clone this lesson from</p> |
| Return value | true if successful |
| Comment s | All data in the target lesson will be deleted/replaced |

| Method | bool Lesson::SetInputName(uint idx, const string& in name) |
|--------------|--|
| Purpose | Set one of the input column names of the lesson |
| Parameters | idx 0-based index of the input to set the name of name String to be set as name |
| Return value | true if successful |
| Comments | None |

| Method | bool Lesson::SetOutputName(uint idx, const string& in name) |
|--------------|---|
| Purpose | Set one of the output column names of the lesson |
| Parameters | idx 0-based index of the output to set the name of name String to be set as name |
| Return value | true if successful |
| Comments | None |

| Method | bool Lesson::GetInputName(uint idx, string& out name) |
|--------------|--|
| Purpose | Get one of the input column names of the lesson |
| Parameters | idx 0-based index of the input to get the name of name Reference to string to be filled with the name |
| Return value | true if successful |
| Comments | None |

| Method | bool Lesson::GetOutputName(uint idx, string& out name) |
|------------|---|
| Purpose | Get one of the output column names of the lesson |
| Parameters | idx 0-based index of the output to get the name of name Reference to string to be filled with the name |

| | |
|--------------|--------------------|
| Return value | true if successful |
| Comments | None |

| | |
|--------------|---|
| Method | bool Lesson::SetInput(uint patternIdx, uint inIdx, double value) |
| Purpose | Set one of the input values of the lesson |
| Parameters | patternIdx 0-based index of the pattern (row in the lesson) to set the value for inIdx 0-based index of the input column to set the value for value The value to be set |
| Return value | true if successful |
| Comments | None |

| | |
|--------------|---|
| Method | bool Lesson::SetOutput(uint patternIdx, uint outIdx, double value) |
| Purpose | Set one of the output values of the lesson |
| Parameters | patternIdx 0-based index of the pattern (row in the lesson) to set the value for outIdx 0-based index of the output column to set the value for value The value to be set |
| Return value | true if successful |
| Comments | None |

| | |
|--------------|---|
| Method | bool Lesson::GetInput(uint patternIdx, uint inIdx, double& out value) |
| Purpose | Get one of the input values of the lesson |
| Parameters | patternIdx 0-based index of the pattern (row in the lesson) to get the value for inIdx 0-based index of the input column to get the value for value Reference to variable to store the value in |
| Return value | true if successful |

| | |
|----------|------|
| Comments | None |
|----------|------|

| | |
|--------------|---|
| Method | bool Lesson::GetOutput(uint patternIdx, uint outIdx, double& out value) |
| Purpose | Get one of the output values of the lesson |
| Parameters | patternIdx 0-based index of the pattern (row in the lesson) to get the value for outIdx 0-based index of the output column to get the value for value Reference to variable to store the value in |
| Return value | true if successful |
| Comments | None |

| | |
|--------------|--|
| Method | void Lesson::SetName(const string& in name) |
| Purpose | Set the name of the lesson |
| Parameters | name The name to be set |
| Return value | None |
| Comments | None |

| | |
|--------------|--|
| Method | void Lesson::SetComment(const string& in comment) |
| Purpose | Set the name of the lesson |
| Parameters | name The name to be set |
| Return value | None |
| Comments | None |

| | |
|--------------|---------------------------------|
| Method | string Lesson::GetName() |
| Purpose | Get the name of the lesson |
| Parameters | None |
| Return value | The name of the lesson |
| Comments | None |

| | |
|---|--|
| s | |
|---|--|

| | |
|--------------|------------------------------------|
| Method | string Lesson::GetComment() |
| Purpose | Get the comment of the lesson |
| Parameters | None |
| Return value | The comment of the lesson |
| Comments | None |

| | |
|--------------|---|
| Method | void Lesson::EnableOutData(bool enable) |
| Purpose | Enables/disables the output data of the lesson |
| Parameters | enable true = enable outputs, false = disable outputs |
| Return value | None |
| Comments | By default every newly created Lesson object has its output data enabled. |

| | |
|--------------|--|
| Method | bool Lesson::IsOutDataEnabled() |
| Purpose | Check if the output data of the lesson is enabled or not |
| Parameters | None |
| Return value | true if enabled |
| Comments | None |

| | |
|--------------|---|
| Method | bool Lesson::DeletePattern(uint idx, uint count) |
| Purpose | Delete a given range of patterns (rows) in the lesson |
| Parameters | idx 0-based index to start deleting from count Number of patterns (rows) to delete. Optional parameter, if omitted then a value of 1 is used |
| Return value | true if successful |
| Comments | None |

| | |
|--------------|---|
| Method | bool Lesson::PatternFromNet(uint idx) |
| Purpose | Read the activations from the currently loaded net's input and output neurons and assign them to the given pattern in the lesson. |
| Parameters | idx 0-based index of the pattern to copy the data to |
| Return value | true if successful |
| Comments | None |

| | |
|--------------|---|
| Method | bool Lesson::AddPatternFromNet() |
| Purpose | Read the activations from the currently loaded net's input and output neurons and assign them to a newly created pattern at the end/tail of the lesson. |
| Parameters | None |
| Return value | true if successful |
| Comments | None |

| | |
|--------------|--|
| Method | void Lesson::Clear() |
| Purpose | Delete all patterns (rows) from the lesson |
| Parameters | None |
| Return value | None |
| Comments | None |

| | |
|------------|---|
| Method | bool Lesson::CreateFftLesson(Lesson@ target, bool complex, bool inputAreColumns, uint minFreqIdx, uint maxFreqPoints) |
| Purpose | Create a new lesson containing the FFT (Fast Fourier Transform) of the input data in the lesson. |
| Parameters | <p>target Target lesson for the FFT, i.e. the Lesson object to receive the FFT data</p> <p>complex If true then a complex value FFT is generated. If false then a real (absolute) value FFT is generated</p> <p>inputAreColumns If true then the input time series patterns for the FFT calculations are taken from the columns of the active lesson. If false then the input patterns are taken from the rows of the currently active lesson.</p> |

| | |
|--------------|---|
| | <p>minFreqIdx (Optional parameter, may be omitted) If this parameter is given then it specifies the minimum frequency index that will appear in the new lesson. Can be used to prevent lower frequencies from appearing in the new lesson.</p> <p>maxFreqPoints (Optional parameter, may be omitted) If this parameter is given then it specifies the maximum number of frequency points to be included in the new lesson starting from 'minFreqIdx'. Note that when this parameter is given then the minFreqIdx parameter must be provided, too.</p> |
| Return value | true if successful |
| Comments | For more details on the behaviour of these functions see section 'FFT Calculations' in the Lesson Editor. |

| | |
|--------------|--|
| Method | bool Lesson::CreateInputAverageLesson(Lesson@ target, uint newInputCount) |
| Purpose | Create a new lesson with a reduced number of inputs by using averaging of the inputs of the lesson. |
| Parameters | <p>target Target lesson for the averaging, i.e. the Lesson object to receive the averaged lesson data</p> <p>newInputCount The number of inputs that shall be present in the target lesson after execution of the method</p> |
| Return value | true if successful |
| Comments | For more details on the behaviour of these functions see section 'Averaging Inputs' in the Lesson Editor. |

| | |
|------------|--|
| Method | bool Lesson::ReadInputsFromImageFile(const string& in fileName, uint resampleCols, uint resampleRows, uint patternIdx, uint inputStartIdx, double grayDark, double grayBright) |
| Purpose | <p>Read a picture into the inputs of the indicated pattern of the lesson as gray scale values. Reading will stop if all pixels have been read or if no more lesson inputs are available.</p> <p>Before reading the image is re-sampled according to the given column and row counts.</p> |
| Parameters | <p>fileName The image file to read the picture from</p> <p>resampleCols Number of columns to apply for re-sampling. If 0 is provided then the number of columns in the original picture file are used.</p> <p>resampleRows</p> |

| | |
|--------------|---|
| | <p>Number of rows to apply for re-sampling. If 0 is provided then the number of rows in the original picture file are used.</p> <p>patternIdx 0-based index of the pattern in the lesson to read into.</p> <p>inputStartIdx (optional) 0-based index into the input array to start reading to. This will be the input position where the pixel in the upper left corner of the image is read to. Default value is 0 if parameter is not provided.</p> <p>grayDark (optional) Value used to represent dark pixels after read-in. Default value is 0 if parameter is not provided.</p> <p>grayDark (optional) Value used to represent bright pixels after read-in. Default value is 1 if parameter is not provided.</p> |
| Return value | true if successful |
| Comments | A large variety of picture file formats is supported, like bmp, jpg, gif, png, mng, ico, tif, tga, pcx, wbmp, wmf, jp2, jpc, pgx, ras, pnm |

| | |
|--------------|---|
| Method | void Lesson::RemovePwd() |
| Purpose | Disables password protection for the lesson. |
| Parameters | None |
| Return value | None |
| Comments | Disable password protection before to save a lesson to disk if you want the file to be stored without password protection. You only need this command in case you already had a password assigned to the lesson object. |

| | |
|--------------|--|
| Method | bool Lesson::SetPwd(const string& in pwd) |
| Purpose | Set the password to be used for loading or saving the lesson. |
| Parameters | <p>pwd The password to be used for this lesson for all Load and Save operations.</p> |
| Return value | true if successful |
| Comments | <p>When a lesson is loaded from file then the provided password only is of effect in case the loaded file has been detected to be encrypted. Else the password is automatically removed from the lesson object after loading the lesson from file.</p> <p>The length of the provided password must be between 1 and 32 characters.</p> |

Teaching

All script commands with respect to teaching the net are described in this chapter.

| | |
|------------|--|
| Command | void SetTeachSpeed(uint msBetweenSteps) |
| Purpose | Adjust the teach speed of MemBrain |
| Parameters | msBetweenSteps (in ms, must be ≥ 0). |
| Comments | The delay time determines for how long MemBrain pauses execution between teaching two patterns of a lesson. I.e. a value of 0 results in the fastest possible teach procedure. |

| | |
|------------|--|
| Command | void SelectTeacher(const string &in teacherName) |
| Purpose | Select the active teacher in the Teacher Manager. |
| Parameters | teacherName |
| Comments | Activates a certain teacher by name according to the Teacher Manager's teacher list. If the list contains more than one teacher with the same name then the first one is picked. If the teacher cannot be found in the Teacher Manager's list then the command fails. Several other commands operate on the currently active teacher. |

| | |
|--------------|--|
| Command | uint GetTeacherCount() |
| Purpose | Returns the number of teachers administered by the Teacher Manager |
| Parameters | none |
| Return value | Current number of teachers in the Teacher Manager |
| Comments | none |

| | | | | | |
|-------------------|---|-------------------|---|-------------|--|
| Command | bool GetTeacherName(uint teacherNum, string &out name) | | | | |
| Purpose | Get the name of a teacher according to its number in the Teacher Manager. | | | | |
| Parameters | <table border="1"> <tr> <td>teacherNum</td><td>Number of the teacher. Must always be > 0 and \leq GetTeacherCount(). The top most teacher in the Teacher Manager has the number 1.</td></tr> <tr> <td>name</td><td>string variable receiving the name if successful</td></tr> </table> | teacherNum | Number of the teacher. Must always be > 0 and \leq GetTeacherCount(). The top most teacher in the Teacher Manager has the number 1. | name | string variable receiving the name if successful |
| teacherNum | Number of the teacher. Must always be > 0 and \leq GetTeacherCount(). The top most teacher in the Teacher Manager has the number 1. | | | | |
| name | string variable receiving the name if successful | | | | |
| Return value | true if successful | | | | |
| Comments | none | | | | |

| | |
|---------|---|
| Command | void TeacherSetting(ETeachSetting setting, double value) void TeacherSetting(ETeachSetting setting, uint value) void TeacherSetting(ETeachSetting setting, EPatternSelect patternSelectMethod) |
|---------|---|

| | | | |
|--|--|---|---|
| Purpose | Set an attribute of the currently active teacher. Note that there are three versions of this function available, depending on the teacher setting that shall be set. | | |
| Parameters Must always be a multiple of two: A setting name followed by a corresponding value | Setting Name | Value | Function |
| | LEARNRATE | double | Set the learnrate of the teacher |
| | TARGET_ERR | double | Set the target error of the teacher |
| | LESSON_REPEATS | uint | Set number of lesson repetitions (must be ≥ 1) |
| | PATTERN_REPEATS | uint | Set number of pattern repetitions (must be ≥ 1) |
| | PATTERN_SELECT | EPatternSelect value: ORDERED or RAND_SELECT or RAND_ORDER | Set the pattern selection method for teaching. |
| Comments | This command always operates on the currently active teacher only (see command SelectTeacher()). Other teachers remain untouched. The new settings are stored with the active teacher so you can change the teacher (or even exit MemBrain and re-start it) and then select the teacher again and it will still have the new settings. | | |

| | | |
|------------|---|--|
| Command | void TeachStep(uint stepCount, bool checkNormRange) | |
| Purpose | Perform a number of Teach Steps of the currently active lesson according to the settings of the currently active teacher. | |
| Parameters | stepCount (Optional Parameter) | The number of Teach Steps to be performed. If this parameter is omitted then one Teach Step is performed. |
| | checkNormRange (Optional Parameter) | If set to true then MemBrain checks that the currently active lesson does not contain any patterns which are out of the specified normalization range for the input or output neurons of the net. If a normalization range violation is detected then the function automatically displays a corresponding warning to the user and lets the user decide to abort the script or to continue i.e. to ignore the warning. The default value is true. I.e. if the function is called without the parameter the function acts as if a parameter of 'true' was handed over. |
| Comments | The script does not wait until the teach process has been performed! If you want to pause the script execution until end of the teach process then use command SleepExec(). | |

| | |
|--|--|
| | Note that the term 'one Teach Step' does not mean that the lesson is only trained one time. This depends on the settings 'Repetitions per Lesson' and 'Repetitions per Pattern' of the active teacher! |
|--|--|

| | | |
|------------|---|---|
| Command | void StartTeaching(bool checkNormRange) | |
| Purpose | Starts the auto teach process using the currently active teacher. | |
| Parameters | checkNormRange (Optional Parameter) | <p>If set to true then MemBrain checks that the currently active lesson does not contain any patterns which are out of the specified normalization range for the input or output neurons of the net.</p> <p>If a normalization range violation is detected then the function automatically displays a corresponding warning to the user and lets the user decide to abort the script or to continue i.e. to ignore the warning.</p> <p>The default value is true. I.e. if the function is called without the parameter the function acts as if a parameter of 'true' was handed over.</p> |
| Comments | <p>The script does not wait until the teach process has been performed!</p> <p>If you want to pause the script execution until end of the teach process then use command SleepExec().</p> | |

| | | |
|------------|---|--|
| Command | void StopTeaching() | |
| Purpose | Stops the currently running teach process | |
| Parameters | None | |
| Comments | MemBrain will stop the teach process and pause execution of the script until the teach process has been stopped completely. | |

| | | |
|--------------|---|--|
| Command | ETeachResult GetLastTeachResult() | |
| Purpose | Returns the result of the last completed teaching procedure. | |
| Parameters | none | |
| Return value | <p>Result of the last teaching procedure. Always one of the following.</p> <p>TR_OK - Normal return value after a single teach step. Also returned if the user stopped the teacher manually.</p> <p>TR_TARGET_NET_ERROR_REACHED - The teacher detected that the target net error has been reached during teaching.</p> <p>TR_MAX_NEURONS_ADDED - The teacher reports that the maximum number of neurons has been added without the target net error being reached - Currently only supported by Cascade Correlation teacher.</p> | |

| | |
|----------|--|
| | TR_TEACH_ABORTED - The teach process has been aborted by the user manually. |
| Comments | In case of errors that lead to automatic abortion of the teach process MemBrain will abort the script execution anyway. Thus, for these error conditions no teach result value is defined for evaluation in scripts. |

| | |
|------------|---|
| Command | uint GetLessonReps() |
| Purpose | Get the counter for lesson repetitions performed by the teacher. |
| Parameters | None |
| Comments | The counter can be reset by the command ResetLessonReps(). The counter is automatically reset in the following situations. - If a new net is created - If a net is loaded from file. - If the net is randomized |

| | |
|------------|---|
| Command | void ResetLessonReps() |
| Purpose | Reset the counter for the number of lesson repetitions performed by the teacher. |
| Parameters | None |
| Comments | The counter is automatically reset in the following situations. - If a new net is created - If a net is loaded from file. - If the net is randomized |

| | |
|--------------|---|
| Method | bool ConfigureNetErrorFunction(EPattErrSummation pattSummationFunction, ENetErrFunc netErrFunction, double binaryThreshold, bool useNormalizedValues) |
| Purpose | Configures the used net error function of MemBrain |
| Parameters | <p>EPattErrSummation One of the following: PATT_SUM_SQUARED PATT_SUM_BINARY_BY_THRESH PATT_SUM_USR</p> <p>ENetErrFunc One of the following: NET_ERR_MEAN NET_ERR_ROOT_MEAN NET_ERR_IDENTITY NET_ERR_USR</p> <p>binaryThreshold Optional parameter (threshold for setting PATT_SUM_BINARY_BY_THRESH)</p> <p>useNormalizedValues Optional parameter (if true then normalized neuron activations are used for error calculation)</p> |
| Return value | true if successful |

| | |
|----------|---|
| Comments | See corresponding menu function |
|----------|---|

Thinking

All script commands with respect to think procedures of the net are described in this chapter.

| | |
|------------|--|
| Command | void SetThinkSpeed(uint msBetweenSteps) |
| Purpose | Adjust the think speed of MemBrain |
| Parameters | msBetweenSteps (in ms, must be ≥ 0) |
| Comments | The delay time determines for how long MemBrain pauses execution between two think steps. I.e. a value of 0 results in the fastest possible think process. |

| | |
|------------|--|
| Command | void ThinkSteps(uint count) |
| Purpose | Perform a number of think (simulation) steps |
| Parameters | count (must be > 0) |
| Comments | The script does not wait until the requested think steps have been performed! If you want to pause the script execution until end of the think steps then use command SleepExec(). |

| | | |
|------------|---|---|
| Command | void ThinkLesson(bool checkNormRange, bool renameWinner) | |
| Purpose | Think on the active lesson: Applies every pattern of the currently active lesson once and performs a think step on it. Always starts with the first pattern in the lesson and ends with the last one. | |
| Parameters | checkNormRange (Optional Parameter) | <p>If set to true then MemBrain checks that the currently active lesson does not contain any patterns which are out of the specified normalization range for the input or output neurons of the net.</p> <p>If a normalization range violation is detected then the function automatically displays a corresponding warning to the user and lets the user decide to abort the script or to continue i.e. to ignore the warning.</p> <p>The default value is true. I.e. if the function is called without the parameter the function acts as if a parameter of 'true' was handed over.</p> |
| | renameWinner (Optional Parameter) | <p>If set to true then MemBrain renames the winner neuron for every pattern in the lesson according to the name of the pattern in the lesson.</p> <p>The default value is false. I.e. if the</p> |

| | | |
|----------|---|--|
| | | function is called without the parameter the function acts as if a parameter of 'false' was handed over. |
| Comments | The script does not wait until the requested operation has been performed! If you want to pause the script execution until end of the Think On Lesson procedure then use command SleepExec(). | |

| | | |
|------------|--|---|
| Command | void ThinkLesson(const Lesson@ less, bool checkNormRange, bool renameWinner) | |
| Purpose | Think on the given Lesson : Applies every pattern of the given Lesson object once and performs a think step on it. Always starts with the first pattern in the lesson and ends with the last one. | |
| Parameters | less | Reference to the Lesson object to use for the command |
| | checkNormRange (Optional Parameter) | <p>If set to true then MemBrain checks that the currently active lesson does not contain any patterns which are out of the specified normalization range for the input or output neurons of the net.</p> <p>If a normalization range violation is detected then the function automatically displays a corresponding warning to the user and lets the user decide to abort the script or to continue i.e. to ignore the warning.</p> <p>The default value is true. I.e. if the function is called without the parameter the function acts as if a parameter of 'true' was handed over.</p> |
| | renameWinner (Optional Parameter) | <p>If set to true then MemBrain renames the winner neuron for every pattern in the lesson according to the name of the pattern in the lesson.</p> <p>The default value is false. I.e. if the function is called without the parameter the function acts as if a parameter of 'false' was handed over.</p> |
| Comments | <p>The script does not wait until the requested operation has been performed! If you want to pause the script execution until end of the Think On Lesson procedure then use command SleepExec().</p> <p>Attention: Ensure that the lifetime of the Lesson object lasts long enough to complete the Think procedure and that the Lesson object is not modified until the Think procedure has ended! E.g. use global lesson object in the script.</p> | |

| | |
|------------|---|
| Command | void StartThink() |
| Purpose | Start the auto think (continuous simulation) mode |
| Parameters | None |

| | |
|----------|--|
| Comments | MemBrain will continue script processing immediately, it does not wait for the think process to be terminated again. If you want to pause the script execution until end of the think procedure (e.g. until the user stops it or until it is stopped remotely over the weblink) then use command SleepExec(). |
|----------|--|

| | |
|------------|---|
| Command | void StopThink() |
| Purpose | Stops the auto think (continuous simulation) mode |
| Parameters | None |
| Comments | MemBrain will stop the think process and pause execution of the script until the think process has been stopped completely. |

| | |
|------------|---|
| Command | void ResetThinkSteps() |
| Purpose | Resets the think step counter on MemBrain's main window |
| Parameters | None |
| Comments | This has no impact on the think procedure itself. The think step counter is for informational purpose only. |

| | |
|--------------|--|
| Command | uint GetOutputWinnerNeuron() |
| Purpose | Get the last know output winner neuron, i.e. the output neuron with the highest (normalized) activation. |
| Parameters | None |
| Return value | Number of the output winner neuron. If valid, the return value is > 0. If no winner neuron is currently defined then the return value is 0. |
| Comments | The top left output neuron has the number 1. Other output neurons have increasing numbers assigned in lines from top to bottom and from left to right. |

Adjusting the View

This section describes all script commands that deal with adjusting MemBrain's view settings.

| | | | |
|--|--|-----------------------------|--|
| Command | void ViewSetting(EViewSetting setting, bool on) | | |
| Purpose | Set an attribute of MemBrain's view. | | |
| Parameters Must always be a multiple of two: A setting name followed by a corresponding value | Setting Name | Value | Function |
| | UPDATE_TEACH | true or false | Enables/Disables update of view during teach |
| | UPDATE_THINK | true or false | Enables/Disables update of view during think |
| | SHOW_LINKS | true or false | Shows/hides links between neurons |

| | | | |
|----------|------------------------|----------------------|---|
| | SHOW_ACT_SPIKES | true or false | Shows/hides activation spikes on links |
| | SHOW_FIRE | true or false | Shows/hides fire indicators on neuron outputs |
| | SHOW_GRID | true or false | Shows/hides the grid |
| | BLACK_BG | true or false | Enables/disables the black background of the window |
| | SHOW_WINNER | true or false | Shows/hides the current winner output neuron |
| Comments | none | | |

| | |
|------------|--|
| Command | void ZoomFit() |
| Purpose | Adjust the zoom of the view so that the whole net is visible |
| Parameters | None |
| Comments | <i>None</i> |

| | |
|------------|----------------------|
| Command | void ZoomIn() |
| Purpose | Zoom in one step |
| Parameters | None |
| Comments | <i>None</i> |

| | |
|------------|-----------------------|
| Command | void ZoomOut() |
| Purpose | Zoom out one step |
| Parameters | None |
| Comments | <i>None</i> |

| | |
|------------|---|
| Command | void EnableScreenUpdate(bool enable) |
| Purpose | Enable/Disable update of the main neural net view during script execution. Can be used to increase performance in case intermediate screen updates are not required. |
| Parameters | enable if true then screen update is enabled. If false then screen update is disabled |
| Comments | <i>None</i> |

Controlling the Weblink

All script commands influencing the Weblink are specified here.

| | | |
|------------|--|---|
| Command | void ActivateWeblink(bool activate) | |
| Purpose | Activate or deactivate the Weblink | |
| Parameters | activate | true activates, false deactivates the Weblink |
| Comments | None | |

Communication with the User

This section handles all aspects of scripting where communication with the user is concerned.

| | | |
|------------|---|--|
| Command | void MessageBox(const string &in message) | |
| Purpose | Display a simple message box with an OK button. | |
| Parameters | message - The text to be displayed in the message box | |
| Comments | <p>Script execution is halted until the user clicks the OK button which closes the message box automatically.</p> <p>Note that the captions on the button is taken from Windows locale settings which means that it is displayed in the same language as the Windows installation</p> | |

| | | |
|--------------|---|--|
| Command | EDlgRet MessageBox(const string &in message, EMsgBoxType type) | |
| Purpose | Display a simple message box with an OK button. | |
| Parameters | <p>message The text to be displayed in the message box</p> <p>type Specifies the type of message box to be displayed. Must be one of the following.</p> <p>MB_OK - Just an OK button</p> <p>MB_OKCANCEL - An OK and a CANCEL button</p> <p>MB_YESNO - YES and NO button</p> <p>MB_YESNOCANCEL - YES, NO and CANCEL button</p> <p>MB_RETRYCANCEL - RETRY and CANCEL button</p> <p>MB_ABORTRETRYIGNORE - ABORT, RETRY and IGNORE button</p> | |
| Return value | <p>Returns information about which button on the message box the user clicked. Always one of the following which corresponds directly to the clicked button.</p> <p>IDOK</p> <p>IDCANCEL</p> | |

| | |
|----------|---|
| | IDYES IDNO IDRETRY IDIGNORE IDABORT |
| Comments | <p>Script execution is halted until the user clicks one of the buttons which closes the message box automatically.</p> <p>Note that the captions on the buttons are taken from Windows locale settings which means that they are displayed in the same language as the Windows installation</p> |

| | |
|--------------|--|
| Command | EDlgRet UserInput(const string &in explanation, double &in initValue, double &out userValue) EDlgRet UserInput(const string &in explanation, int &in initValue, int &out userValue), EDlgRet UserInput(const string &in explanation, const string &in initValue, string &out userValue), (Three versions of this function which support the data types 'double', 'string' and 'int'. All three versions behave in the same way). |
| Purpose | Request the user to enter a value. Different versions of the function support different data types (see overloaded function versions above). |
| Parameters | <p>explanation The text to be displayed to the user. Typically an explanation on the type of data that shall be entered and what its meaning is. If "" is provided as explanation (empty string) then a default explanation is provided that will identify the requested data type.</p> <p>initValue Specifies the initial value in the data entry field that is displayed to the user</p> <p>userValue Variable that received the actual user input. Only valid if the user has clicked the OK button (return value IDOK).</p> |
| Return value | <p>Returns information about which button on the user input dialog box the user clicked. Always one of the following which corresponds directly to the clicked button.</p> <p>IDOK</p> <p>IDCANCEL</p> |
| Comments | <p>Script execution is halted until the user clicks one of the buttons which closes the dialog box automatically.</p> <p>Note that the captions on the buttons are taken from Windows locale settings which means that they are displayed in the same language as the Windows installation</p> |

| | |
|--------------|--|
| Command | EDlgRet FileOpenDlg(const string &in title, const string &in extension, const string &in fileNameInit, string &out fileName) |
| Purpose | Open a file selection dialog to let the user choose a file for being opened. Get the user selected file name back from the function. |
| Parameters | <p>title The title to be displayed on the dialog</p> <p>extension Specifies the default extension that shall be used to browse files in the dialog. E.g. the extension string "txt" sets the file filtering to all files with extension "txt". Provide "*" or just "" (i.e. an empty string) if you want to set no file filter. The dialog will in this case browse all file types (*.*);</p> <p>fileNameInit You can provide an initial file name here to be displayed in the dialog. Provide "" if you don't want to specify any initial file name.</p> <p>fileName The full file name (including path) which the user selected. Only valid if the function returns with the value IDOK.</p> |
| Return value | <p>Returns information about which button on the user input dialog box the user clicked. Always one of the following which corresponds directly to the clicked button.</p> <p>IDOK</p> <p>IDCANCEL</p> |
| Comments | <p>Script execution is halted until the user clicks one of the buttons which closes the dialog box automatically.</p> <p>Note that the captions on the buttons are taken from Windows locale settings which means that they are displayed in the same language as the Windows installation</p> |

| | |
|------------|--|
| Command | EDlgRet FileSaveDlg(const string &in title, const string &in extension, const string &in fileNameInit, string &out fileName) |
| Purpose | Open a file selection dialog to let the user choose a file for being saved to. Get the user selected file name back from the function. |
| Parameters | <p>title The title to be displayed on the dialog</p> <p>extension Specifies the default extension that shall be used to browse files in the dialog. E.g. the extension string "txt" sets the file filtering to all files with extension "txt". Provide "*" or just "" (i.e. an empty string) if you want to set no file filter. The dialog will in this case browse all file types (*.*);</p> <p>fileNameInit You can provide an initial file name here to be displayed in the dialog. Provide "" if you don't want to specify any initial file name.</p> <p>fileName The full file name (including path) which the user selected.</p> |

| | |
|--------------|---|
| | Only valid if the function returns with the value IDOK. |
| Return value | Returns information about which button on the user input dialog box the user clicked. Always one of the following which corresponds directly to the clicked button. IDOK IDCANCEL |
| Comments | Script execution is halted until the user clicks one of the buttons which closes the dialog box automatically. Note that the captions on the buttons are taken from Windows locale settings which means that they are displayed in the same language as the Windows installation |

| | |
|------------|--|
| Command | void ShowTraceWin(bool show) |
| Purpose | Show/Hide the scripting trace window |
| Parameters | show - 'true' shows the trace window, 'false' hides it. |
| Comments | A script can put text messages to the tracing window using the script command 'Trace(...)' |

| | |
|------------|---|
| Command | void Trace(const string &in text) |
| Purpose | Output a text string to the scripting trace window |
| Parameters | text - The text to output to the tracing window. |
| Comments | Append "\r\n" to the text if you want the next text be added to a new line in the trace window. If the tracing window has never been shown since starting MemBrain it will be shown automatically when this command is executed. When the trace window is closed by the user, however, then it will not automatically get shown! |

| | |
|------------|-----------------------------------|
| Command | void ClearTraceWin() |
| Purpose | Clear the scripting trace window. |
| Parameters | none |
| Comments | none |

Controlling External Applications

This section covers all script commands and elements that deal with controlling external applications.

| | |
|----------|--|
| Function | uint64 StartApp(const string &in fileName, const string &in params) |
| Purpose | Start an external application and get a handle to it. |

| | | |
|--------------|--|---|
| Parameters | fileName | The full path to the external application that shall be started |
| | params | The command line parameter string that is passed to the external application. |
| Return value | The handle to the application if successful. Else 0. | |
| Comments | The script should always check the return value to be > 0 to ensure that the operation was successful. If the script shall be able to make reference to the running application later on then the returned handle must be stored in some variable. | |

| | | |
|------------|--|-------------------------------------|
| Function | void SleepAppExit(uint64 appHandle) | |
| Purpose | Pause the script execution until an external application has terminated | |
| Parameters | appHandle | Handle to the external application. |
| Comments | An application handle value is returned by the script function 'StartApp(...)' when an external application has been started successfully. The script has to keep the handle in some variable to make reference to the external application after starting it. | |

| | | |
|--------------|--|--|
| Function | bool SleepAppExit(uint64 appHandle, uint maxTimeMs) | |
| Purpose | Pause the script execution until an external application has terminated | |
| Parameters | appHandle | Handle to the external application. |
| | maxTimeMs | Maximum time to wait for the application to terminate. |
| Return value | true if the function returns because the application has terminated or if the handle value is invalid. false if the function returns because of timeout. | |
| Comments | An application handle value is returned by the script function 'StartApp(...)' when an external application has been started successfully. The script has to keep the handle in some variable to make reference to the external application after starting it. | |

| | | |
|--------------|--|-------------------------------------|
| Function | bool IsAppRunning(uint64 appHandle) | |
| Purpose | Check if an external application is still running | |
| Parameters | appHandle | Handle to the external application. |
| Return value | true if the application is running. false if the application has terminated or the handle is invalid. | |
| Comments | An application handle value is returned by the script function 'StartApp(...)' when an external application has been started successfully. The script has to keep the handle in some variable to make reference to the external application after starting it. | |

| | | |
|----------|--|--|
| Function | | |
|----------|--|--|

| Function | bool TerminateApp(uint64 appHandle, uint exitCode) | |
|--------------|--|---|
| Purpose | Terminate an external application with a given exit code | |
| Parameters | appHandle | Handle to the external application that shall be terminated. |
| | exitCode | The exit code the external application shall return to the system when exiting. |
| Return value | true on success. false if the handle is invalid. | |
| Comments | An application handle value is returned by the script function 'StartApp(...)' when an external application has been started successfully. The script has to keep the handle in some variable to make reference to the external application after starting it. | |

| Function | bool GetAppExitCode(uint64 appHandle, uint &out exitCode) | |
|--------------|--|---|
| Purpose | Get the exit code of an external application | |
| Parameters | appHandle | Handle to the external application for which the exit code shall be returned. |
| | exitCode | The exit code the external application returned to the system when exiting. |
| Return value | true on success. false if the handle is invalid or if the external application is still running. | |
| Comments | An application handle value is returned by the script function 'StartApp(...)' when an external application has been started successfully. The script has to keep the handle in some variable to make reference to the external application after starting it. | |

| Function | bool ShellExecute(const string &in operation, const string &in fileName, const string &in params) | |
|------------|--|---|
| Purpose | Execute a Windows Shell command. | |
| Parameters | operation | One of the following strings. "open" The operation opens the file specified by the fileName parameter. The file can be an executable file or a document file. It can also be a folder. "print" The operation prints the file specified by fileName. The file should be a document file. If the file is an executable file, the function opens the file, as if "open" had been specified. "explore" The function explores the folder specified by fileName in Windows Explorer. |
| | fileName | The full path to the file, application or folder that the operation shall be performed with. |
| | params | The command line parameter string that is |

| | | |
|--------------|------------------|--|
| | | passed to an application if the operation 'open' is performed. |
| Return value | true on success. | |
| Comments | none | |

Stock Management

This section covers all script commands and elements that deal with controlling MemBrain's [Neural Net Stock](#).

| | | |
|--------------|--|--|
| Function | bool GetCaptureBestNetOnStock() | |
| Purpose | Get option flag if the best net shall be captured on stock during teaching or not | |
| Parameters | None | |
| Return value | true if the option is active, false if it is inactive | |
| Comments | This option is also available via MemBrain's main menu <Teach><Capture Best Net on Stock...>. A check mark besides the menu entry indicates that the option is active. | |

| | | |
|------------|---|--|
| Command | void SetCaptureBestNetOnStock(bool capture) | |
| Purpose | Activate or deactivate the option flag to capture the best net on stock during teaching. | |
| Parameters | capture | true activates, false deactivates the option |
| Comments | This option is also available via MemBrain's main menu <Teach><Capture Best Net on Stock...>. A check mark besides the menu entry indicates that the option is active. Clicking on the menu entry opens a dialog that allows to toggle the option activation state. | |

| | | |
|--------------|---|---|
| Command | ENetComparisonMethod GetNetComparisonMethod(double &out maxDevPercent) | |
| Purpose | Get the comparison method used to identify the best net during teaching for the purpose of capturing it on the stock. | |
| Parameters | maxDevPercent (optional parameter) | Max percentage deviation between net error and train error. This value takes effect in combination with NET_COMP_MEAN_TRAIN_NET_ERR_MAX_DEV only. |
| Return value | Currently used net comparison method. One of the following values which correspond to the selection in the available dialog in MemBrain. <ul style="list-style-type: none"> • NET_COMP_NET_ERR • NET_COMP_DIFF_TRAIN_NET_ERR • NET_COMP_MEAN_TRAIN_NET_ERR | |
| Comments | This setting is also available via MemBrain's main menu <Teach><Capture Best Net on Stock...>. A dialog appears that shows | |

| | |
|--|---|
| | the currently selected method and also allows to change it. |
|--|---|

| | | |
|------------|---|---|
| Command | void SetNetComparisonMethod(ENetComparisonMethod method, double maxDevPercent) | |
| Purpose | Set the comparison method used to identify the best net during teaching for the purpose of capturing it on the stock. | |
| Parameters | method | Net comparison method to be used. The following values are available which correspond to the selection in the available dialog in MemBrain. <ul style="list-style-type: none"> • NET_COMP_NET_ERR • NET_COMP_DIFF_TRAIN_NET_ERR • NET_COMP_MEAN_TRAIN_NET_ERR • NET_COMP_MEAN_TRAIN_NET_ERR_MAX_DEV |
| | maxDevPercent (optional parameter) | Max percentage deviation between net error and train error. This value takes effect in combination with NET_COMP_MEAN_TRAIN_NET_ERR_MAX_DEV only. The setting of the value is allowed in combination with all other methods, however. Allowed value range is 0 .. 10000 |
| Comments | This option is also available via MemBrain's main menu <Teach><Capture Best Net on Stock...>. A check mark besides the menu entry indicates that the option is active. Clicking on the menu entry opens a dialog that allows to toggle the option activation state. | |

| | |
|--------------|---|
| Function | uint CaptureNetOnStock() |
| Purpose | Capture the currently edited net on the stock. |
| Parameters | None |
| Return value | The 1-based stock slot where the net has been captured on the stock |
| Comments | The command will always create a new stock slot for the capturing. Default values for name and documentation of the captured net will be generated by MemBrain. |

| | | |
|------------|--|--|
| Command | uint CaptureNetOnStock(const string &in name, const string &in documentation) | |
| Purpose | Capture the currently edited net on the stock with given name and documentation | |
| Parameters | name | The name for the net in the stock |
| | documentation | The documentation for the net in the stock |

| | |
|--------------|---|
| Return value | The 1-based stock slot where the net has been captured on the stock |
| Comments | The command will always create a new stock slot for the capturing. |

| | | |
|--------------|---|--|
| Command | bool ReplaceNetOnStock(uint slot) | |
| Purpose | Capture the currently edited net on the stock by replacing an exiting other net on stock at the given stock slot. | |
| Parameters | slot | The 1-based stock slot where the net shall be captured. Must be an existing slot on the stock. |
| Return value | true if successful, false on error | |
| Comments | Default values for name and documentation of the captured net will be generated by MemBrain. | |

| | | |
|--------------|---|--|
| Command | bool ReplaceNetOnStock(uint slot, const string &in name, const string &in documentation) | |
| Purpose | Capture the currently edited net on the stock by replacing an exiting other net on stock at the given stock slot. | |
| Parameters | slot | The 1-based stock slot where the net shall be captured. Must be an existing slot on the stock. |
| | name | The name for the net in the stock |
| | documentation | The documentation for the net in the stock |
| Return value | true if successful, false on error | |
| Comments | None | |

| | | |
|--------------|--|---|
| Command | bool LoadNetFromStock(uint slot) | |
| Purpose | Load the currently edited net from the stock by providing a slot number | |
| Parameters | slot | The 1-based stock slot the net shall be loaded from. Must be an existing slot on the stock. |
| Return value | true if successful, false on error | |
| Comments | The currently edited net will be replaced by the one from the stock. Note that this action can be manually undone via MemBrain's normal <Undo> operation | |

| | | |
|--------------|---|--|
| Command | bool LoadNetFromStock(const string &in name) | |
| Purpose | Load the currently edited net from the stock by name | |
| Parameters | name | The name of the stock net which shall be loaded. |
| Return value | true if successful, false on error | |
| Comments | The currently edited net will be replaced by the one from the stock. Note that this action can be manually undone via MemBrain's normal <Undo> operation. If multiple nets on the stock exist with this name then always the | |

| | |
|--|---|
| | historically first (oldest) one with this name is loaded! |
|--|---|

| | | |
|--------------|---|---|
| Command | bool SaveStock(const string &in fileName) | |
| Purpose | Save the whole stock to file | |
| Parameters | fileName | The full path of the file to save the stock to. If only a file name is provided then the stock will be saved in the working directory of the script. |
| Return value | true if successful, false on error | |
| Comments | The standard extension for stock files is 'mbs' and it is recommended to use this extension. However, any other extension can be used either. | |

| | | |
|--------------|---|--|
| Command | bool LoadStock(const string &in fileName) | |
| Purpose | Load the whole stock from file | |
| Parameters | fileName | The full path of the file to load the stock from. If only a file name is provided then the stock will be loaded from the working directory of the script. |
| Return value | true if successful, false on error | |
| Comments | The standard extension for stock files is 'mbs' and it is recommended to use this extension. However, any other extension can be used either. | |

| | | |
|--------------|--|--|
| Function | bool ClearStock() | |
| Purpose | Clear the stock (remove all nets from the stock) | |
| Parameters | None | |
| Return value | true if successful, false on error | |
| Comments | The command will fail (return false) in case the option <Teach><Capture Best Net On Stock> is active and teaching is active at the same time. In this case the stock may not be cleared since the teacher of MemBrain relies on a stock slot allocated in the beginning of the teach process to auto-capture the best net during teaching. | |

| | | |
|--------------|--|--|
| Command | uint GetLastTeacherAutoCaptureStockSlot() | |
| Purpose | Get the stock slot used by the last teaching process to auto-capture the best net on stock. | |
| Parameters | None | |
| Return value | The 1-based stock slot number used by the last teaching process | |
| Comments | If after script based teaching the best net shall be re-loaded from stock this function can be used to identify the appropriate stock slot to load the net from. | |

| | |
|--------------|--|
| Command | uint GetStockSize() |
| Purpose | Get the number of nets currently captured on the stock |
| Parameters | None |
| Return value | The number of nets in the stock |
| Comments | None |

Arbitrary File Access

MemBrain's scripting language supports arbitrary access to files both read and write. This section documents the scripting language's file class that implements this functionality. Note that you don't the file class to open or save neural nets or lessons. For these known file types other script functions are readily available. The file class described in this chapter allows to access files in a user defined way both in binary and text mode.

Arbitrary file access is implemented in an object oriented way in MemBrain scripts. I.e. there is a class available that represents a file. The methods of this class perform operations on the file.

The following example shows how a file is being opened for reading and how a line of text is being read from the file.

```
void main()
{
    // Create a file object and open it for reading. Don't deny access to
the file
    // for other applications.
    file myFile;
    if (!myFile.Open("SomeFile.txt", FILE_MODE_READ | FILE_SHARE_DENY_NONE))
    {
        MessageBox("Unable to open file");
    }
    else
    {
        // Read one line of text into the variable 'text' and display it
string text;

        if (myFile.Read(text))
        {
            MessageBox(text);
        }
        else
        {
            MessageBox("Error reading from file!");
        }
    }
}
```

The following paragraphs document all methods available in the file class

| | |
|------------|--|
| Method | bool file::Open(const string &in fileName, uint openFlags) |
| Purpose | Open a file in various modes |
| Parameters | fileName The path and name of the file. Can be also a relative path |

| | |
|--------------|--|
| | <p>openFlags This is a combination of the following flags that have to be combined by bitwise OR operation (' ').</p> <p>mode flags:</p> <p>FILE_MODE_CREATE Causes the function to create a new file. If the file exists already, it is truncated to 0 length.</p> <p>FILE_MODE_NO_TRUNCATE Combine this value with FILE_MODE_CREATE. If the file being created already exists, it is not truncated to 0 length. Thus the file is guaranteed to open, either as a newly created file or as an existing file. This might be useful, for example, when opening a settings file that may or may not exist already.</p> <p>FILE_MODE_READ Opens the file for reading only.</p> <p>FILE_MODE_WRITE Opens the file for writing only.</p> <p>FILE_MODE_READ_WRITE Opens the file for reading and writing.</p> <p>share access flags</p> <p>FILE_SHARE_DENY_NONE Opens the file without denying other processes read or write access to the file.</p> <p>FILE_SHARE_DENY_READ Opens the file and denies other processes read access to the file.</p> <p>FILE_SHARE_DENY_WRITE Opens the file and denies other processes write access to the file.</p> <p>FILE_SHARE_EXCLUSIVE Opens the file with exclusive mode, denying other processes both read and write access to the file.</p> <p>file type flags</p> <p>FILE_TYPE_TEXT Sets text mode with special processing for carriage return–linefeed pairs (default in MemBrain scripts)</p> <p>FILE_TYPE_BINARY Sets binary mode</p> |
| Return value | true if successful |

| | |
|----------|---|
| Comments | The script should always check the return value to be true to ensure that the operation was successful. |
|----------|---|

| | |
|------------|--|
| Method | void file::Close() |
| Purpose | Closes a file |
| Parameters | none |
| Comments | MemBrain will automatically close all open files when the script exits. Still, it is good programming praxis to explicitly close files when they are not needed anymore. |

| | |
|--------------|--|
| Method | bool file::SeekToBegin() |
| Purpose | Move the file access pointer to the beginning of the file. |
| Parameters | none |
| Return value | true if successful |
| Comments | none |

| | |
|--------------|--|
| Method | bool file::SeekToEnd() |
| Purpose | Move the file access pointer to the end of the file. |
| Parameters | none |
| Return value | true if successful |
| Comments | none |

| | | |
|--------------|---|--|
| Method | int file::Seek(int offset, EFilePos fromFilePos) | |
| Purpose | Move the file access pointer by a given offset. | |
| Parameters | offset | Number of bytes the file pointer shall be moved. Use negative values to move the pointer towards the beginning of the file and positive values to move the pointer towards the end of the file. |
| | fromFilePos | The position from which the file pointer shall be moved. Must be always one of the following. FILE_POS_BEGIN Move the pointer to a position of <offset> bytes with respect to the beginning of the file. FILE_POS_CURRENT Move the pointer to a position of <offset> bytes with respect to the current position in file. FILE_POS_END Move the pointer to a position of <offset> bytes with respect to the end of the file. |
| Return value | The new file access position with respect to the beginning of the file. | |

| | |
|----------|---------------------------|
| | < 0 if an error occurred. |
| Comments | none |

| | |
|--------------|--|
| Method | bool file::GetLength(uint &out length) |
| Purpose | Get the length of the file in bytes |
| Parameters | length Variable receiving the length of the file if successful |
| Return value | true if successful |
| Comments | none |

| | |
|--------------|---|
| Method | bool file::Flush() |
| Purpose | Forces all bytes in the file buffer to be written to the physical file immediately. |
| Parameters | none |
| Return value | true if successful |
| Comments | The method Close() automatically flushes the file buffer. Thus, you need the Flush() method only when you want to force writing to a physical file before it is being closed. This can be useful if some other application shall be able to read contents from a file while the file shall be kept open in the MemBrain script. |

| | |
|--------------|---|
| Method | bool file::Read(bool &out value) bool file::Read(uint8 &out value) bool file::Read(int8 &out value) bool file::Read(uint16 &out value) bool file::Read(int16 &out value) bool file::Read(uint &out value) bool file::Read(int &out value) bool file::Read(float &out value) bool file::Read(double &out value) |
| Purpose | Read a simple data type from the file |
| Parameters | value Variable receiving the read data if successful |
| Return value | true if successful |
| Comments | Use FILE_TYPE_BINARY when opening the file in order to read binary data properly in all circumstances! See file::Open(...) for details. |

| | |
|--------------|---|
| Method | bool file::Write(bool value) bool Write(uint8 value) bool Write(int8 value) bool Write(uint16 value) bool Write(int16 value) bool Write(uint value) bool Write(int value) bool Write(float value) bool Write(double value) |
| Purpose | Write a simple data type to the file |
| Parameters | value Value to be written to the file |
| Return value | true if successful |
| Comments | Use FILE_TYPE_BINARY when opening the file in order to write binary data properly in all circumstances! See file::Open(...) for details. |

| | |
|--------------|--|
| Method | uint file::Read(string &out buffer, uint readLen) |
| Purpose | Read a number of bytes from the file into a string that is used as buffer. |
| Parameters | buffer A string buffer that receives the data. readLen The number of bytes to be read from the file |
| Return value | The number of successfully read bytes. |
| Comments | Use FILE_TYPE_BINARY when opening the file in order to read binary data properly in all circumstances! See file::Open(...) for details. Note that the string object used for this method interface is not used as a text string. It can contain any binary data and accessed like an array. For more information on the strings supported in MemBrain scripts see here . |

| | |
|--------------|--|
| Method | bool Write(const string &in buffer, uint writeLen) |
| Purpose | Write a number of bytes from the provided buffer to the file. |
| Parameters | buffer A string object used as byte buffer that contains the data to be written. writeLen The number of bytes to be written to the file |
| Return value | true on success. |

| | |
|----------|---|
| Comments | Use FILE_TYPE_BINARY when opening the file in order to read binary data properly in all circumstances! See <code>file::Open(...)</code> for details. Note that the string object used for this method interface is not used as a text string. It can contain any binary data and accessed like an array. For more information on the strings supported in MemBrain scripts see here . |
|----------|---|

| | |
|--------------|---|
| Method | bool Read(string &out str) |
| Purpose | Read one line from a text file |
| Parameters | str The string object that will receive the text string. |
| Return value | true if successful. |
| Comments | Reading stops on the first newline character that occurs during reading |

| | |
|--------------|--|
| Method | bool Write(const string &in str) |
| Purpose | Write the given text string to the file |
| Parameters | str The text string to be written to the file. |
| Return value | true if successful. |
| Comments | When the file is opened in mode <code>FILE_TYPE_TEXT</code> then every newline character found in <code><str></code> is written to the file as a carriage return / line feed pair. |

Files and Directories

This section deals with the capabilities to create or remove directories, delete existing files and retrieve directory information

| | |
|--------------|---|
| Command | bool MakeDir(string &in dirName) |
| Purpose | Create a new directory |
| Parameters | dirName The name of the new directory |
| Return value | true if successful |
| Comments | If <code><dirName></code> consists of several directories separated with <code>"\"</code> then only the last sub directory of the path may be not existing and will be created. Else the function will fail and return false. The function will return successful if the directory already exists. |

| | |
|--------------|--|
| Command | bool RemoveDir(string &in dirName) |
| Purpose | Remove a directory |
| Parameters | dirName The name of the directory to be removed |
| Return value | true if successful |

| | |
|----------|---|
| Comments | <p>If <dirName> consists of several directories separated with "\\" then only the last sub directory of the path will be removed. Else the function will fail and return false.</p> <p>The directory that shall be removed must be empty.</p> |
|----------|---|

| | | |
|--------------|--|------------------------------------|
| Command | bool DeleteFile(string &in dirName) | |
| Purpose | Delete a file | |
| Parameters | fileName | The name of the file to be deleted |
| Return value | true if successful | |
| Comments | None | |

| | | |
|--------------|--|---|
| Command | void GetWorkDir(string &out workDir) | |
| Purpose | Get the working directory of the currently executed script | |
| Parameters | workDir | String variable that will receive the working directory information |
| Return value | none | |
| Comments | None | |

In addition to the above listed commands there is also a filesystem class available in the scripting language. You can create a file system object and then use its methods to retrieve information about the file system and perform extended operations.

See http://www.angelcode.com/angelscript/sdk/docs/manual/doc_script_stdlib_filesystem.html for more information on the available methods.

Strings

The MemBrain scripting library supports a powerful string class that supports convenient handling of text strings. Additionally, the string class supports simple dynamic arrays of bytes to be transferred to and from [user defined files](#).

The following sections state all methods and operators implemented for the string class. Additionally, there are string utility functions available that operate on strings. See [here](#) for details on these functions.

Constructors

The following table lists all available ways to construct a string object

| | |
|--------------|---|
| Constructors | <p>string @f() Construct an empty string object</p> <p>string @f(const string &in) Construct a new string object as a copy of another string object</p> |
| Comments | None |

Supported operators

The string class supports the following operators.

| | |
|--|---|
| <string> <operation> <string> | Valid operators for operations on two string objects = += == != <= >= + |
| Comments | None |

| | |
|--|---|
| <string> <operation> <double> | Valid operators for operations on a string and a double object = + += |
| Comments | none |

| | |
|---|--|
| <string> <operation> <float> | Valid operators for operations on a string and a float object = + += |
| Comments | none |

| | |
|--|---|
| <string> <operation> <int> | Valid operators for operations on a string and an int object = + += |
| Comments | none |

| | |
|--|---|
| <string> <operation> <uint> | Valid operators for operations on a string and a uint object = + += |
| Comments | none |

Access to single bytes in a string

Use `string[i]` to access the byte with index `i`. Indices in strings always start with 0.

Example:

```
string text = "Hello";
```

```
uint8 character = text[0]; // the variable 'character' is assigned the value 'H'.
```

Methods of the string class

This section lists all available methods of the <string> class.

| Method | uint string::length() const |
|--------------|------------------------------------|
| Purpose | Get the length of the string |
| Parameters | none |
| Return value | The length of the string |
| Comments | none |

| Method | void string::resize(uint newSize) |
|--------------|---|
| Purpose | Set a new length of the string. This can be used to prepare a buffer for storing a byte stream to a file for example. |
| Parameters | newSize The new size (length) of the string |
| Return value | none |
| Comments | none |

| Method | bool string::isEmpty() const |
|--------------|---|
| Purpose | Returns true if the string is empty, i.e. the length is zero. |
| Parameters | none |
| Return value | true if the string is empty |
| Comments | none |

| Method | string@ substr(int startIdx = 0, int length = -1) const |
|--------------|---|
| Purpose | Return a sub string of the string |
| Parameters | startIdx The start index for the sub string count The length of the sub string |
| Return value | The default arguments will return the whole string as the new string. |
| Comments | The default paranme |

| Method | void insert(uint pos, const string &in other) |
|--------------|---|
| Purpose | Inserts another string <i>other</i> at position <i>pos</i> in the original string. |
| Parameters | pos Position index to insert the other string at other The string to insert into this string |
| Return value | none |
| Comments | None |

| | | |
|--------------|--|---|
| Method | void erase(int startIdx, int count = -1) | |
| Purpose | Erases a range of characters from the string, starting at position <i>startIdx</i> and counting <i>count</i> characters. | |
| Parameters | startIdx | The start index for erasing |
| | count | The number if characters to erase from the string |
| Return value | count = -1 will erase all remaining characters from the string | |
| Comments | The default paranme | |

| | | |
|--------------|---|---|
| Method | int findFirst(const string &in subStr, int startIdx) const | |
| Purpose | Find the first occurrence of the value subStr in the string, starting at startIdx. If no occurrence is found a negative value will be returned. | |
| Parameters | subStr | The sub string to search for in the string |
| | startIdx (optional) | The index specifying where to start with the search |
| Return value | The index of the first occurrence of the sub string within the string. -1 if sub string is not found. | |
| Comments | None | |

| | | |
|--------------|--|---|
| Method | int findLast(const string &in subStr, int startIdx=-1) const | |
| Purpose | Find the last occurrence of the value subStr in the string. If startIdx is >= 0 the search will begin at that position, i.e. any potential occurrence after that position will not be searched. If no occurrence is found a negative value will be returned. | |
| Parameters | subStr | The sub string to search for in the string |
| | startIdx (optional) | The index specifying where to start with the search |
| Return value | The index of the last occurrence of the sub string within the string. -1 if sub string is not found. | |
| Comments | None | |

| | | |
|---------|--|--|
| Method | int findFirstOf(const string &in chars, int start = 0) const int findFirstNotOf(const string &in chars, int start = 0) const int findLastOf(const string &in chars, int start = -1) const int findLastNotOf(const string &in chars, int start = -1) const | |
| Purpose | <p>The first variant finds the first character in the string that matches on of the characters in chars, starting at start. If no occurrence is found a negative value will be returned.</p> <p>The second variant finds the first character that doesn't match any of</p> | |

| | | |
|--------------|---|---|
| | those in chars. The third and last variant are the same except they start the search from the end of the string. | |
| Parameters | chars | Character set to search for |
| | start | The index specifying where to start with the search |
| Return value | The index first occurrence of one of the characters in chars. -1 if none of the characters is found. | |
| Comments | These functions work on the individual bytes in the strings. They do not attempt to understand encoded characters, e.g. UTF-8 encoded characters that can take up to 4 bytes. | |

| | | |
|--------------|---|--|
| Method | array<string>@ split(const string &in delim) const | |
| Purpose | <p>This function splits the string it into parts by looking for a specified delimiter. Example:</p> <pre>string str = "A B D"; string[] array = str.split(" ");</pre> <p>The resulting array has the following elements: {"A", "B", "", "D"}</p> | |
| Parameters | str | The string to split into sub strings |
| | delim | The delimiter to use to split the string |
| Return value | An array of the sub strings | |
| Comments | None | |

String Utilities

This section lists a set of string utilities functions that can be used in combinations with strings.

| | | |
|--------------|---|---|
| Function | string@ join(const string@[] &in stringArray, const string &in delim) | |
| Purpose | <p>This function takes as input an array of string handles as well as a delimiter and concatenates the array elements into one delimited string. Example:</p> <pre>string@[] array = {"A", "B", "", "D"}; string str = join(array, " ");</pre> <p>The resulting string is: "A B D"</p> | |
| Parameters | stringArray | The string array to be joined into one delimited string |
| | delim | The delimiter to use to join the sub strings |
| Return value | The joined string | |
| Comments | None | |

| | |
|----------|--|
| Function | int64 parseInt(const string &in str, uint base = 10, uint &out byteCount = 0) |
|----------|--|

| | |
|--------------|---|
| | uint64 parseInt(const string &in str, uint base = 10, uint &out byteCount = 0) |
| Purpose | Parses the string for an integer value. The base can be 10 or 16 to support decimal numbers or hexadecimal numbers. If byteCount is provided it will be set to the number of bytes that were considered as part of the integer value. |
| Parameters | See Purpose |
| Return value | The parsed integer value |
| Comments | None |

| | |
|--------------|---|
| Function | double parseFloat(const string &in, uint &out byteCount = 0) |
| Purpose | Parses the string for a floating point value. If byteCount is provided it will be set to the number of bytes that were considered as part of the value. |
| Parameters | See Purpose |
| Return value | The parsed floating point value |
| Comments | None |

| | |
|--------------|--|
| Function | string formatInt(int64 val, const string &in options = "", uint width = 0) string formatUInt(uint64 val, const string &in options = "", uint width = 0) string formatFloat(double val, const string &in options = "", uint width = 0, uint precision = 0) |
| Purpose | <p>The format functions takes a string that defines how the number should be formatted. The string is a combination of the following characters:</p> <p>l = left justify 0 = pad with zeroes + = always include the sign, even if positive space = add a space in case of positive number h = hexadecimal integer small letters (not valid for formatFloat) H = hexadecimal integer capital letters (not valid for formatFloat) e = exponent character with small e (only valid for formatFloat) E = exponent character with capital E (only valid for formatFloat)</p> |
| Parameters | See Purpose |
| Return value | The formatted string |
| Comments | <p>Examples:</p> <pre>// Left justify number in string with 10 characters string justified = formatInt(number, 'l', 10);</pre> |

| | |
|--|--|
| | <pre>// Create hexadecimal representation with capital letters, right justified string hex = formatInt(number, 'H', 10); // Right justified, padded with zeroes and two digits after decimal separator string num = formatFloat(number, '0', 8, 2);</pre> |
|--|--|

| Function | double StringToDouble(const string &in s) double StringToDouble(const string &in s, const string &in decSep) | |
|--------------|---|---|
| Purpose | Convert a string into its double value representation | |
| Parameters | s | The string to convert |
| | decSep | Optional parameter: string that consists of the decimal separator that shall be used for the conversion. Default value is "." |
| Return value | The value representation of the string as double. | |
| Comments | Legacy function Returns 0 if the string is not convertible into a number | |

| Function | int StringToInt(const string &in s) | |
|--------------|---|-----------------------|
| Purpose | Convert a string into its signed 32 bit integer value representation | |
| Parameters | s | The string to convert |
| Return value | The value representation of the string a signed 32 bit integer. | |
| Comments | Legacy function Returns 0 if the string is not convertible into a number | |

Maths

This section lists all special math related functions available to the scripting language. Note that simple math operations like +, -, /, * or % are not listed here since they are part of the AngelScript library by default anyway.

Trigonometric functions

| Function | double cos(double value) | |
|--------------|---------------------------------|--|
| Purpose | Calculate the cosine of a value | |
| Parameters | value | The value to calculate the cosine from |
| Return value | The cosine of the value | |
| Comments | None | |

| | | |
|--------------|---------------------------------|--------------------------------------|
| Function | double sin(double value) | |
| Purpose | Calculate the sine of a value | |
| Parameters | value | The value to calculate the sine from |
| Return value | The sine of the value | |
| Comments | None | |

| | | |
|--------------|----------------------------------|---|
| Function | double tan(double value) | |
| Purpose | Calculate the tangens of a value | |
| Parameters | value | The value to calculate the tangens from |
| Return value | The tangens of the value | |
| Comments | None | |

| | | |
|--------------|---------------------------------------|--|
| Function | double acos(double value) | |
| Purpose | Calculate the arcus cosine of a value | |
| Parameters | value | The value to calculate the arcus cosine from |
| Return value | The arcus cosine of the value | |
| Comments | None | |

| | | |
|--------------|-------------------------------------|--|
| Function | double asin(double value) | |
| Purpose | Calculate the arcus sine of a value | |
| Parameters | value | The value to calculate the arcus sine from |
| Return value | The arcus sine of the value | |
| Comments | None | |

| | | |
|--------------|--|---|
| Function | double atan(double value) | |
| Purpose | Calculate the arcus tangens of a value | |
| Parameters | value | The value to calculate the arcus tangens from |
| Return value | The arcus tangens of the value | |
| Comments | None | |

Hyberbolic functions

| | | |
|----------|----------------------------------|--|
| Function | double cosh(double value) | |
|----------|----------------------------------|--|

| | |
|--------------|--|
| | |
| Purpose | Calculate the hyperbolic cosine of a value |
| Parameters | value The value to calculate the hyperbolic cosine from |
| Return value | The hyperbolic cosine of the value |
| Comments | None |

| | |
|--------------|--|
| Function | double sinh(double value) |
| Purpose | Calculate the hyperbolic sine of a value |
| Parameters | value The value to calculate the hyperbolic sine from |
| Return value | The hyperbolic sine of the value |
| Comments | None |

| | |
|--------------|---|
| Function | double tanh(double value) |
| Purpose | Calculate the hyperbolic tangens of a value |
| Parameters | value The value to calculate the hyperbolic tangens from |
| Return value | The hyperbolic tangens of the value |
| Comments | None |

Exponential and logarithmic functions

| | |
|--------------|--|
| Function | double log(double value) |
| Purpose | Calculate the natural logarithm of a value |
| Parameters | value The value to calculate the natural logarithm from |
| Return value | The natural logarithm of the value |
| Comments | None |

| | |
|--------------|--|
| Function | double log10(double value) |
| Purpose | Calculate the base 10 logarithm of a value |
| Parameters | value The value to calculate the base 10 logarithm from |
| Return value | The base 10 logarithm of the value |
| Comments | None |

Power functions

| | | |
|--------------|--|------------------------------|
| Function | double pow(double first, double second) | |
| Purpose | Returns of value of the first argument raised to the power of the second argument. | |
| Parameters | first | See description in 'Purpose' |
| | second | |
| Return value | The result of the calculation | |
| Comments | None | |

| Function | double sqrt(double value) | |
|--------------|--------------------------------------|---|
| Purpose | Calculate the square root of a value | |
| Parameters | value | The value to calculate the square root from |
| Return value | The square root of the value | |
| Comments | None | |

Nearest integer, absolute value, and remainder functions

| Function | double ceil(double value) | |
|--------------|---|-------------------------------------|
| Purpose | Calculate the the smallest (closest to negative infinity) double value that is not less than the argument and is equal to a mathematical integer. | |
| Parameters | value | The value to calculate the ceil for |
| Return value | The ceil of the value | |
| Comments | None | |

| Function | double abs(double value) | |
|--------------|---|--|
| Purpose | Returns the absolute value of a double value. If the argument is not negative, the argument is returned. If the argument is negative, the negation of the argument is returned. | |
| Parameters | value | The value to return the absolute value for |
| Return value | The absolute value of the value | |
| Comments | None | |

| Function | double floor(double value) | |
|----------|---|--|
| Purpose | Calculate the the the largest (closest to positive infinity) double value that is not greater than the argument and is equal to a mathematical integer. | |

| | | |
|--------------|------------------------|--------------------------------------|
| Parameters | value | The value to calculate the floor for |
| Return value | The floor of the value | |
| Comments | None | |

| | | |
|--------------|---|---|
| Function | double fraction(double value) | |
| Purpose | This function returns the signed fractional portion of <value>. | |
| Parameters | value | The value to calculate the fraction for |
| Return value | The fraction of the value | |
| Comments | Example: fraction(-34.6754) returns the value -0.6754 | |

| | | |
|--------------|---|--|
| Function | double Random() | |
| Purpose | Get a pseudo random value between 0 and 1 | |
| Parameters | none | |
| Return value | Pseudo random value between 0 and 1 | |
| Comments | None | |

| | | |
|--------------|---|------------------------------------|
| Function | float fpFromIEEE(uint value) | |
| Purpose | This function converts a uint value to an IEEE 754 float value. | |
| Parameters | value | The value to be converted to float |
| Return value | The value as floating point representation | |
| Comments | None | |

| | | |
|--------------|---|-----------------------------------|
| Function | uint fpToIEEE(float value) | |
| Purpose | This function converts an IEEE 754 float value to a uint value. | |
| Parameters | value | The value to be converted to uint |
| Return value | The value as uint representation | |
| Comments | None | |

| | | |
|----------|--|--|
| Function | double fpFromIEEE(uint64 value) | |
| Purpose | This function converts a uint64 value to an IEEE 754 double value. | |

| | | |
|--------------|--|-------------------------------------|
| Parameters | value | The value to be converted to double |
| Return value | The value as floating point representation | |
| Comments | None | |

| | | |
|--------------|--|-------------------------------------|
| Function | uint64 fpToIEEE(double value) | |
| Purpose | This function converts an IEEE 754 double value to a uint64 value. | |
| Parameters | value | The value to be converted to uint64 |
| Return value | The value as uint64 representation | |
| Comments | None | |

FFT Calculations

This section deals with the capabilities to calculate FFTs (Fast Fourier Transformations) through scripting.

| | | |
|--------------|---|---|
| Command | bool CreateFftLesson(bool complex, bool inputAreColumns, uint minFreqIdx, uint maxFreqPoints) | |
| Purpose | Create a new lesson containing the FFT (Fast Fourier Transform) of the input data in the currently active lesson. | |
| Parameters | complex | If true then a complex value FFT is generated. If false then a real (absolute) value FFT is generated |
| | inputAreColumns | If true then the input time series patterns for the FFT calculations are taken from the columns of the active lesson. If false then the input patterns are taken from the rows of the currently active lesson. |
| | minFreqIdx | (Optional parameter, may be omitted) If this parameter is given then it specifies the minimum frequency index that will appear in the new lesson. Can be used to prevent lower frequencies from appearing in the new lesson. |
| | maxFreqPoints | (Optional parameter, may be omitted) If this parameter is given then it specifies the maximum number of frequency points to be included in the new lesson starting from 'minFreqIdx'. Note that when this parameter is given then the minFreqIdx parameter must be provided, too. |
| Return value | true if successful | |
| Comments | For more details on the behaviour of these functions see section 'FFT Calculations' in the Lesson Editor. | |

| | | |
|---------|---|--|
| Command | double GetFftFrequency(uint freqIdx, double overallSampleTime) | |
| Purpose | Get the absolute frequency of an FFT component entry specified by 0-based index.. | |

| | | |
|--------------|---|--|
| Parameters | freqIdx | The 0-based index of the FFT component the frequency shall be calculated for. |
| | overallSampleTime | The overall sample time (in seconds) contained in one time series sample that was the input for the corresponding FFT. |
| Return value | The absolute frequency value of the FFT entry. If an error occurs (i.e. if overallSampleTime <= 0) then a value < 0 is returned. | |
| Comments | For more details on the behaviour of these functions see section 'FFT Calculations' in the Lesson Editor. | |

Serial Ports

The scripting language of MemBrain provides the class **SerialPort** to access the serial (COM) ports of your computer.

Create an object of the SerialPort class for every COM-port that you want to access.

Constructors

The following table lists all available ways to construct a SerialPort object.

| | |
|--------------|--|
| Constructors | <p>SerialPort @f() Construct a default SerialPort object. Communication parameters will be: COM-Port: COM1 Baud rate: 9600 Data bits: 8 Parity: None Stop bits: 1 No hardware handshake</p> <p>SerialPort @f(uint16 comPortNum) Construct a new SerialPort object for the given COM port. All other parameters will be initialized as with the default constructor</p> |
| Comments | None |

| | |
|--------------|---|
| Method | void SerialPort::SetPort(uint16 comPortNum) |
| Purpose | Set the COM port number to be used. Use this function BEFORE to open the port! |
| Parameters | comPortNum The number of the used COM port (e.g. 1 for COM1) |
| Return value | None |
| Comments | You will only need this function if you want to set or change the used COM port after construction. Note that if the currently assigned COM port is already open it will be closed in case you change it to a different value using this function. |

| | |
|--------|--------------------------------|
| Method | bool SerialPort::Open() |
|--------|--------------------------------|

| | |
|--------------|-------------------|
| Purpose | Open the COM port |
| Parameters | None |
| Return value | true on success. |
| Comments | None |

| | |
|--------------|--|
| Method | bool SerialPort::Setup(EBaudRate baud, EDataBits dataBits, EParity parity, EStopBits stopBits) |
| Purpose | Setup the parameters for the used COM port. Use this function AFTER the COM port has been opened! |
| Parameters | <p>baud The baud rate as one of the following.</p> <p>BAUD_110 BAUD_300 BAUD_600 BAUD_1200 BAUD_2400 BAUD_4800 BAUD_9600 BAUD_14400 BAUD_19200 BAUD_38400 BAUD_56000 BAUD_57600 BAUD_115200 BAUD_128000 BAUD_256000</p> <p>dataBits The number of data bits used as one of the following.</p> <p>DATA_BITS_5 DATA_BITS_6 DATA_BITS_7 DATA_BITS_8</p> <p>parity The parity setting as one of the following.</p> <p>PARITY_NONE PARITY_ODD PARITY_EVEN PARITY_MARK PARITY_SPACE</p> <p>stopBits</p> <p>STOP_BITS_1 STOP_BITS_1_5 STOP_BITS_2</p> |
| Return value | true on success |
| Comments | Do not use this function before the COM port has been opened |

| | |
|--|---|
| | successfully. Else It will fail and also cause the open function to fail! |
|--|---|

| | |
|--------------|--|
| Method | bool SerialPort::SetupHandshaking(EHandshake handshake) |
| Purpose | Setup the handshaking method for the used COM port. Use this function AFTER the COM port has been opened! |
| Parameters | handshake The handshaking method as one of the following. HANDSHAKE_OFF HANDSHAKE_HARDWARE HANDSHAKE_SOFTWARE |
| Return value | true on success |
| Comments | Do not use this function before the COM port has been opened successfully. Else It will fail and also cause the open function to fail! |

| | |
|--------------|---------------------------------|
| Method | void SerialPort::Close() |
| Purpose | Close the COM port |
| Parameters | None |
| Return value | None |
| Comments | None |

| | |
|--------------|--|
| Method | uint SerialPort::Read(string &out buffer, uint readLen) |
| Purpose | Read a number of bytes from the COM port into a string that is used as buffer. |
| Parameters | buffer A string buffer that receives the data. readLen The number of bytes to be read from the COM ports receive buffer. |
| Return value | The number of successfully read bytes. |
| Comments | The function does not wait for the specified number of bytes being received, i.e. if there are less bytes available than specified to be read then the function immediately returns with only the already received bytes read. |

| | |
|--------------|---|
| Method | bool SerialPort::WaitRx(uint timeoutMs) |
| Purpose | Wait for data being received by the COM port. |
| Parameters | timeoutMs The maximum time to wait for RX data getting available on the COM port. |
| Return value | true if data is available to read from the port. |
| Comments | If there is already data available in the receive buffer of the port then |

| | |
|--|---|
| | the function will return true immediately and not wait for further data to be received. |
|--|---|

| | |
|--------------|--|
| Method | void SerialPort::ClearRx() |
| Purpose | Clear the receive buffer of the COM port and discard any data received so far. |
| Parameters | None |
| Return value | None |
| Comments | None |

| | |
|--------------|--|
| Method | bool SerialPort::Write(const string &in buffer, uint writeLen) |
| Purpose | Write a number of bytes from the provided buffer to the COM port. |
| Parameters | buffer A string object used as byte buffer that contains the data to be written. writeLen The number of bytes to be written to the COM port |
| Return value | true on success. |
| Comments | None |

| | |
|--------------|--|
| Method | uint SerialPort::Read(string &out buffer) |
| Purpose | Read all available received data from the COM port into the provided buffer. |
| Parameters | buffer The string buffer object that will receive the read data. |
| Return value | Number of read bytes which were written to the buffer. |
| Comments | None |


| | |
|--------------|--|
| Method | bool SerialPort::Write(string &out buffer) |
| Purpose | Write all data in the given buffer to the COM port |
| Parameters | buffer The string buffer that contains the data to be written. |
| Return value | true if successful. |
| Comments | None |

Debugging Scripts

MemBrain comes with a powerful external source code level debugger for scripts.

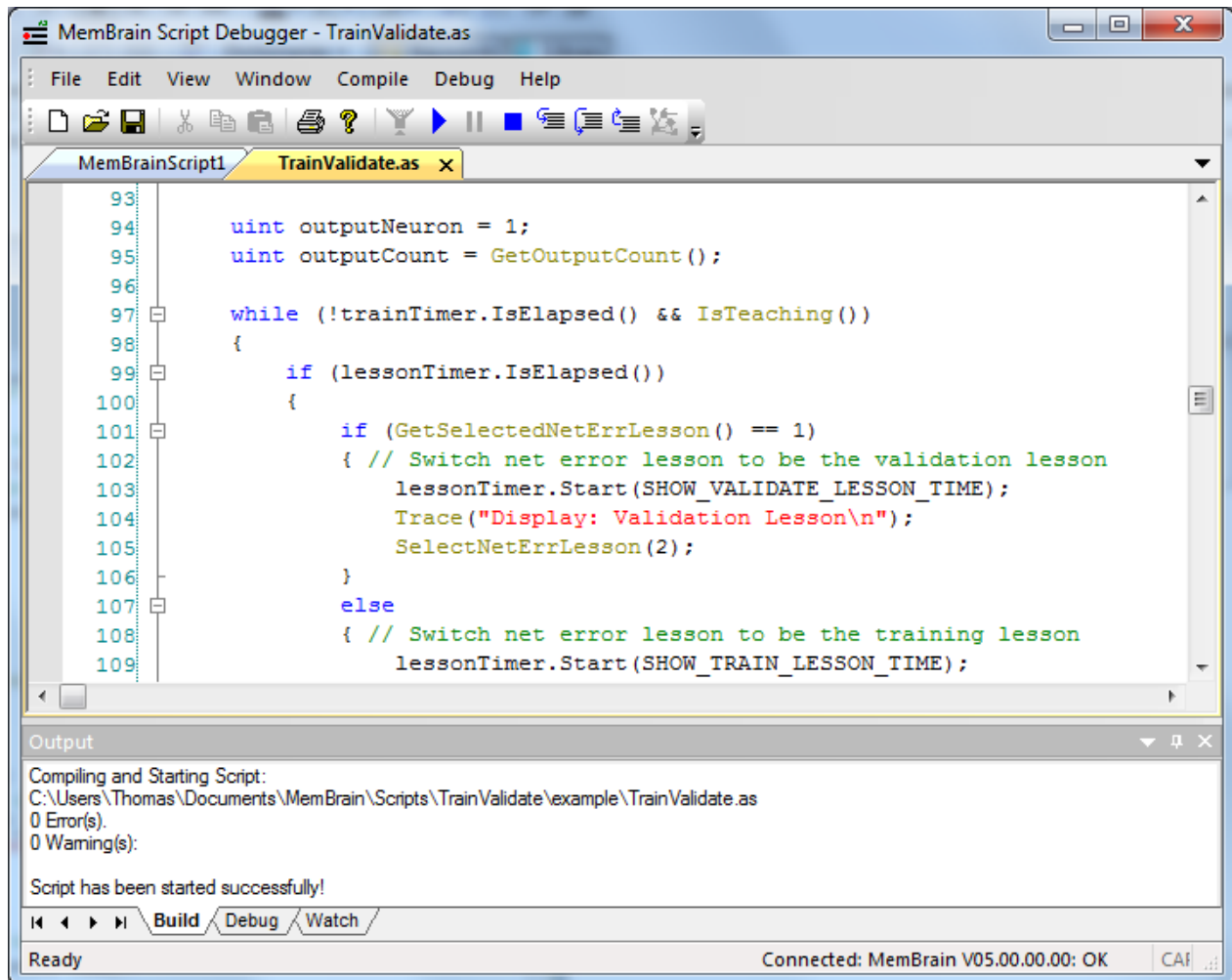
Note: Do not use the bracket characters '(' or ')' in the file names of your MemBrain scripts. The 'jump to code' feature of the debugger won't work with these since the bracket characters are used to parse the code locations to jump to when double-clicking on a debugger compiler message.

The debugger can be started via the following alternative ways:

- Click on the tool bar icon 
- Select the menu command <Scripting><Launch Script Debugger>
- Start the debugger via the standard Windows Programs (located in the same program group as MemBrain itself).

Note: If a script is currently running in MemBrain when the debugger is started then the debugger will automatically attach to the running script and halt the script at its current source code location.

The script debugger looks like following:



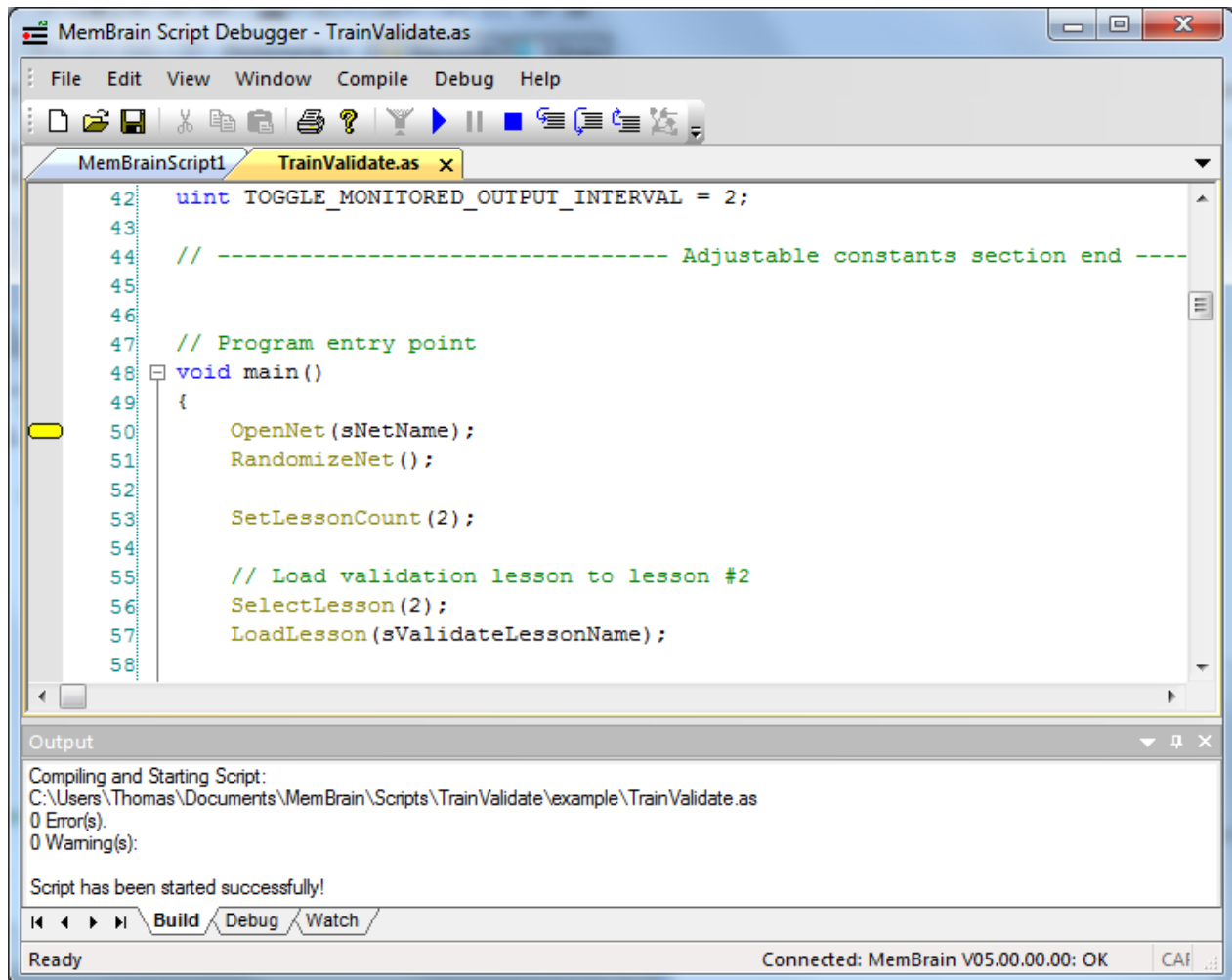
The debugger supports the following features:

- Syntax highlighting
 - AngelScript Language keywords: **blue**
 - MemBrain keywords and function names: **ochre**
 - Comments: **green**
 - Text constants: **red**
 - Other text: **black**
- Fully featured text editor, including block indentation, copy/paste, search function, bookmarks...
- Compile, build run and pause script remotely from within debugger
- Build output window: See compiler messages directly in the debugger window, jump to code location via mouse click
- Debug output window

- Variable watch window
- Graphical breakpoints
- Step Into, Step Over, Step Out
- Attach debugger to running script
- Run debugger on the same machine as MemBrain or on an other machine within the LAN
- Start debugger from within MemBrain
- Start MemBrain from within debugger

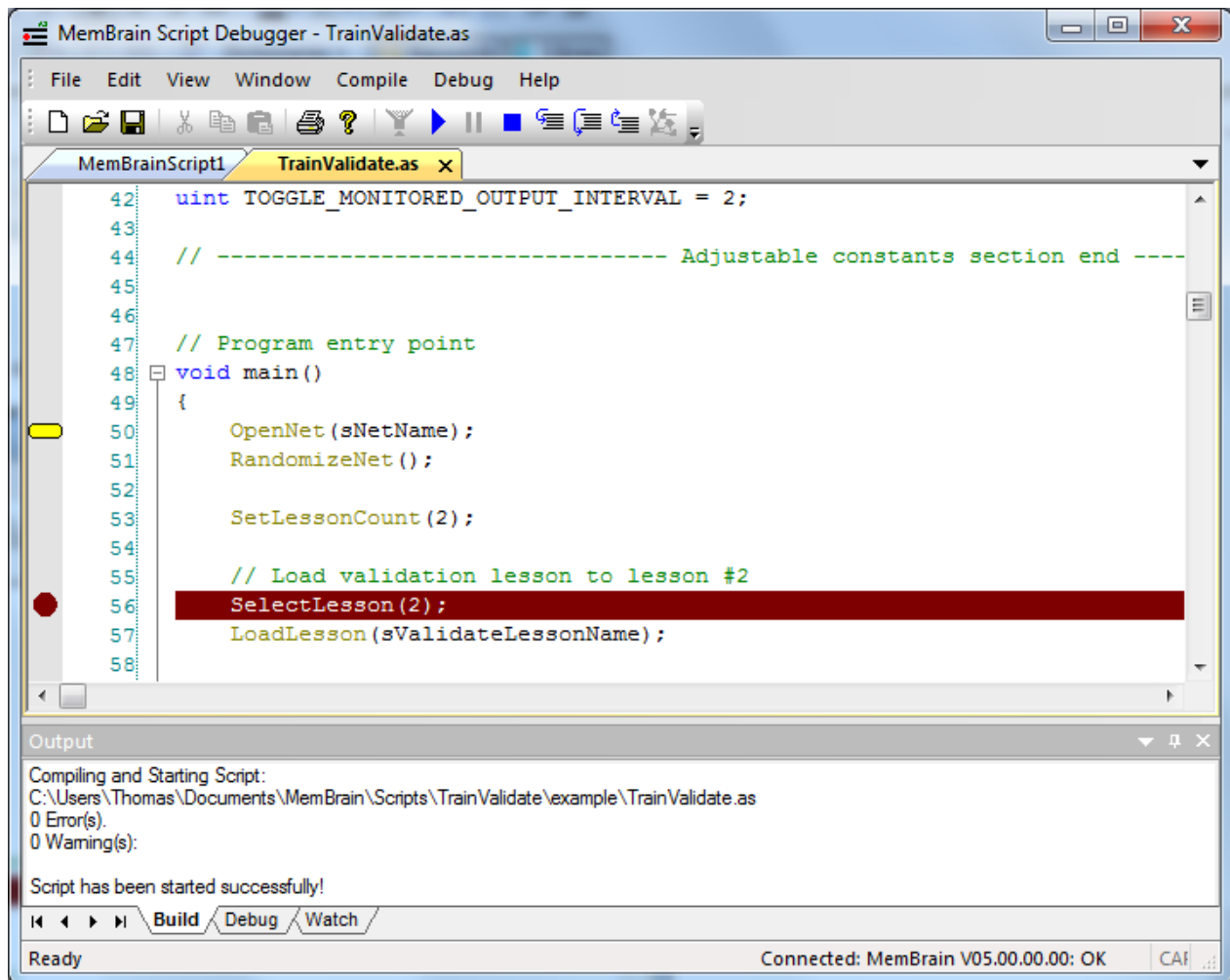
Execution Position Marker

When the debugger is halted and also immediately after starting a script in the debugger the current execution position in the script code is identified via a yellow marker on the left side of the source code:



Breakpoints

Breakpoints in the code are identified by red bullet-shaped markers on the left side of the source code:



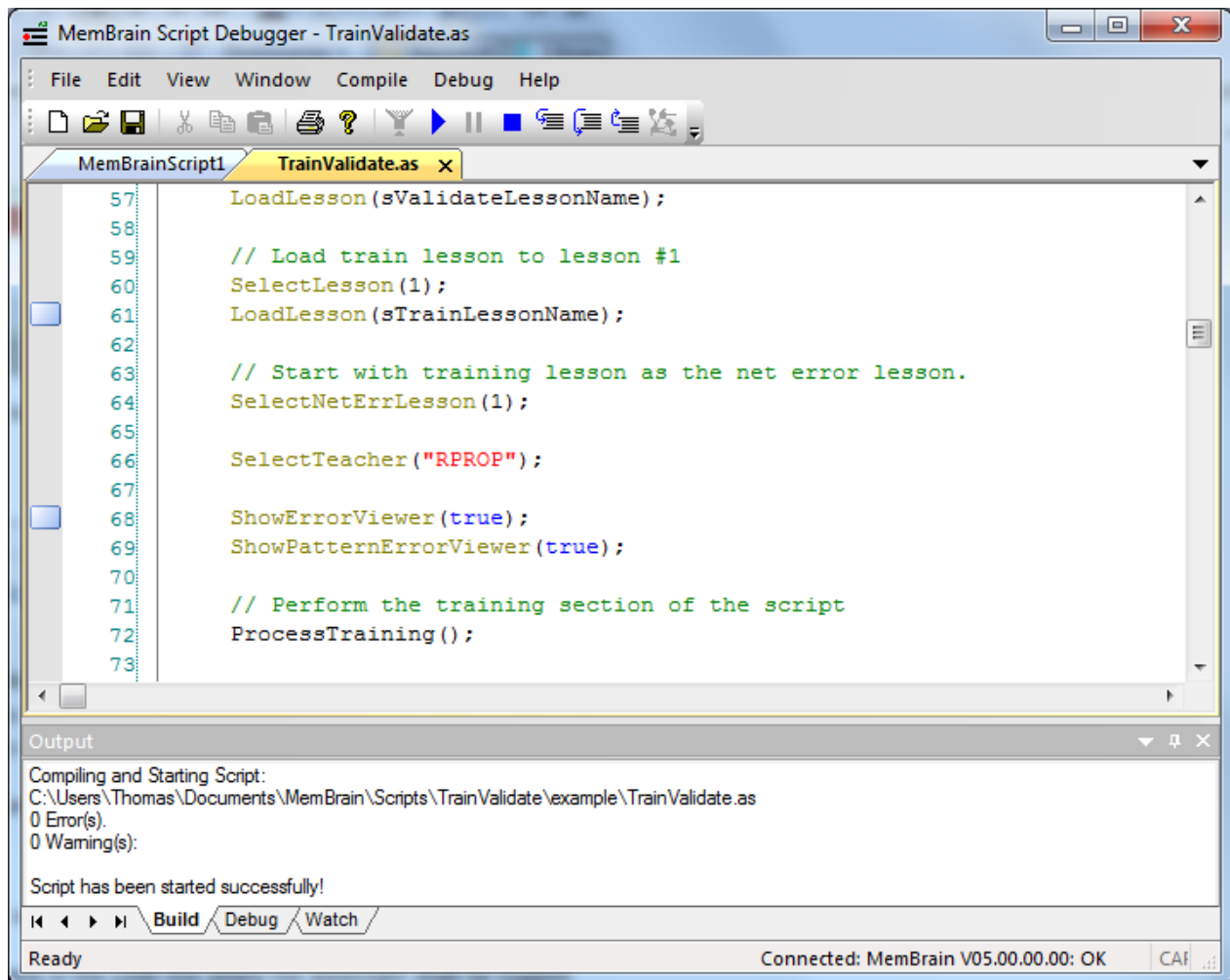
Breakpoints can be toggled by:

- Double-clicking in the area on the left side of the source code where the breakpoints are displayed
- Hitting F9 on the keyboard while the cursor is somewhere in the code line where the breakpoint shall be toggled.
- Selecting the menu option <Debug><Insert/Remove Breakpoint>

All breakpoints can be removed via the menu option <Debug><Remove All Breakpoints>

Bookmarks

Bookmarks in the code can be used to remember code locations in a file and jump to them. They are displayed on the left side of the source code area and are represented by light-blue markers as in the following picture.



A bookmark can be toggled (set/removed) via the following actions:


- Hitting <Ctrl> + F2 on the keyboard while the cursor is somewhere in the code line where the bookmark shall be toggled.
- Selecting the menu option <Edit><Toggle Bookmark>

Navigation to the next/previous bookmark is done via:


- Hitting F2 or <Shift> + F2 on the keyboard, respectively
- Selecting the menu options <Edit><Next Bookmark> or <Previous Bookmark>, respectively

Compiling a script and identifying compile errors

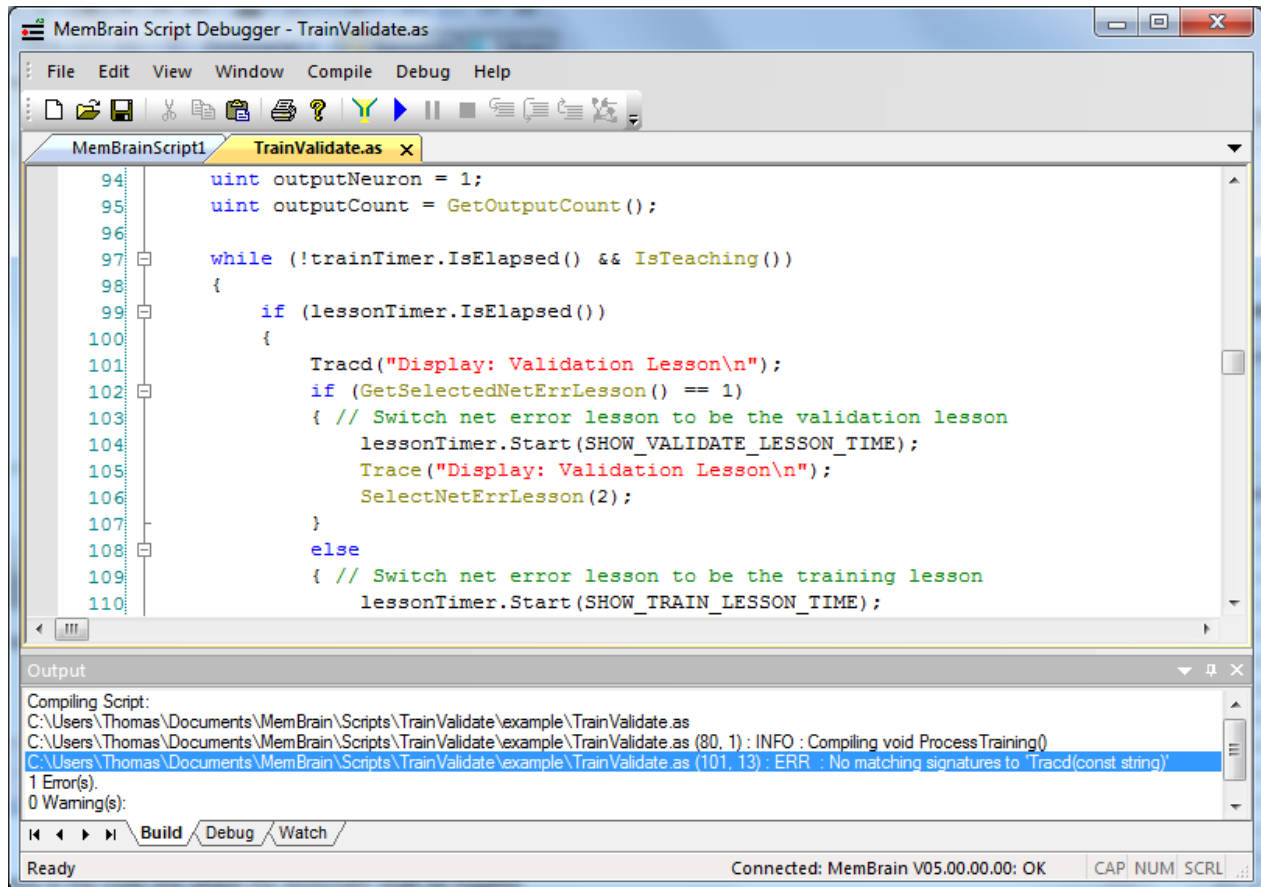
To compile a script without immediately starting it use either of the following options.

Note: MemBrain must be running in order to compile scripts and the status line of the debugger window must indicate a successful connection to MemBrain. If MemBrain is not yet running you can either start it manually or start MemBrain from within the debugger using the toolbar icon .

Alternatively, select the menu command <Debug><Launch MemBrain>. Also, the script file must be saved to disk before compilation. If the file has not yet been saved the debugger will request the save operation before compilation starts.

- Click the toolbar icon 
- Press F7 on the keyboard
- Select the menu item <Compile><Compile Script>


The Build Window will monitor all build messages from MemBrain including warnings and errors like in the following example:



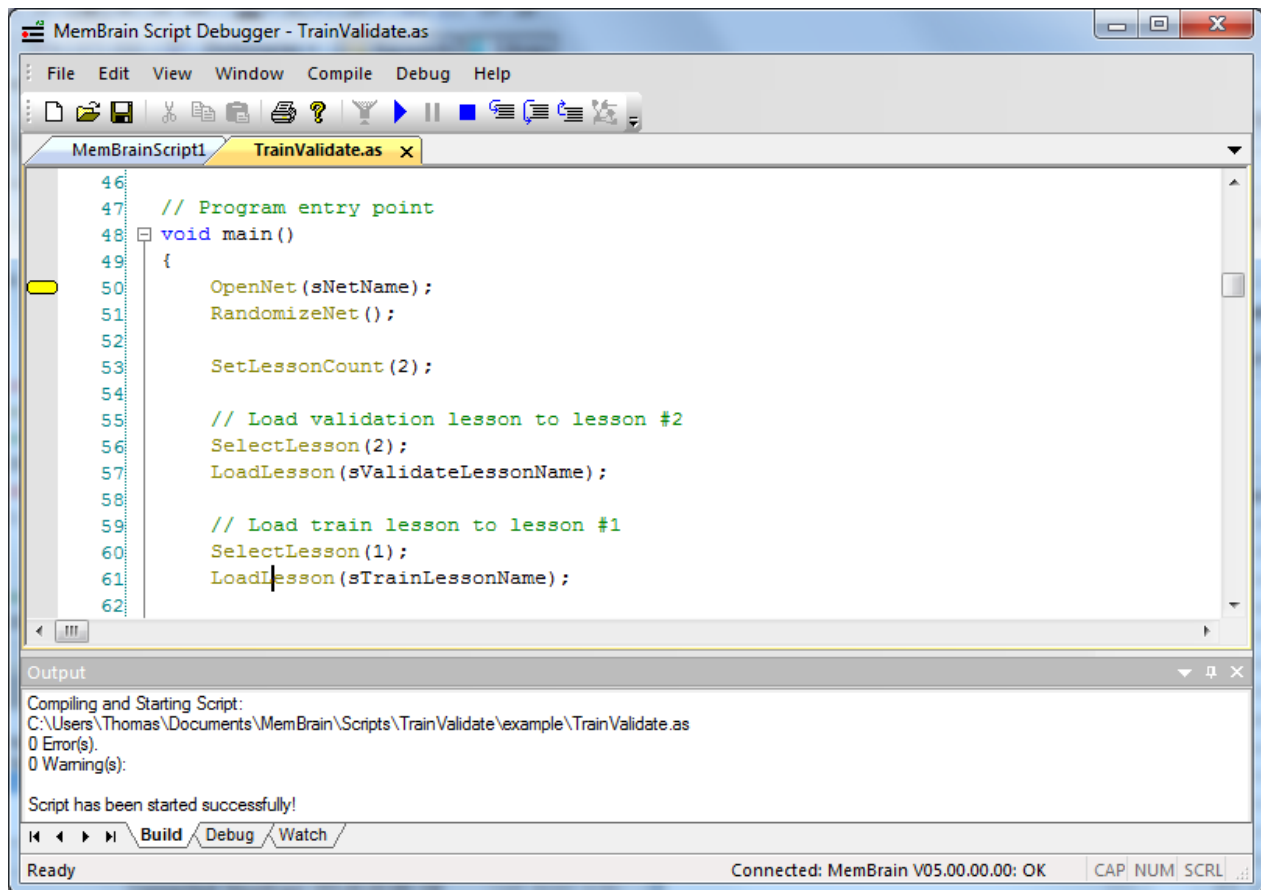
A double-click on the error message will directly take your cursor to the code location which caused the error (in this example the MemBrain command 'Trace' in line 101 contains a typo: It is typed 'Tracd'. Also note that the syntax highlighting shows that this is not a known keyword. Compare the correctly typed 'Trace' command in line 105 which is printed with syntax highlighting indicating that it is a known MemBrain command.

Running a script

In order to run a script from within the debugger perform one of the following:

- Click the toolbar icon 
- Hit F5 on the keyboard
- Select the menu item <Debug><Run/Continue>

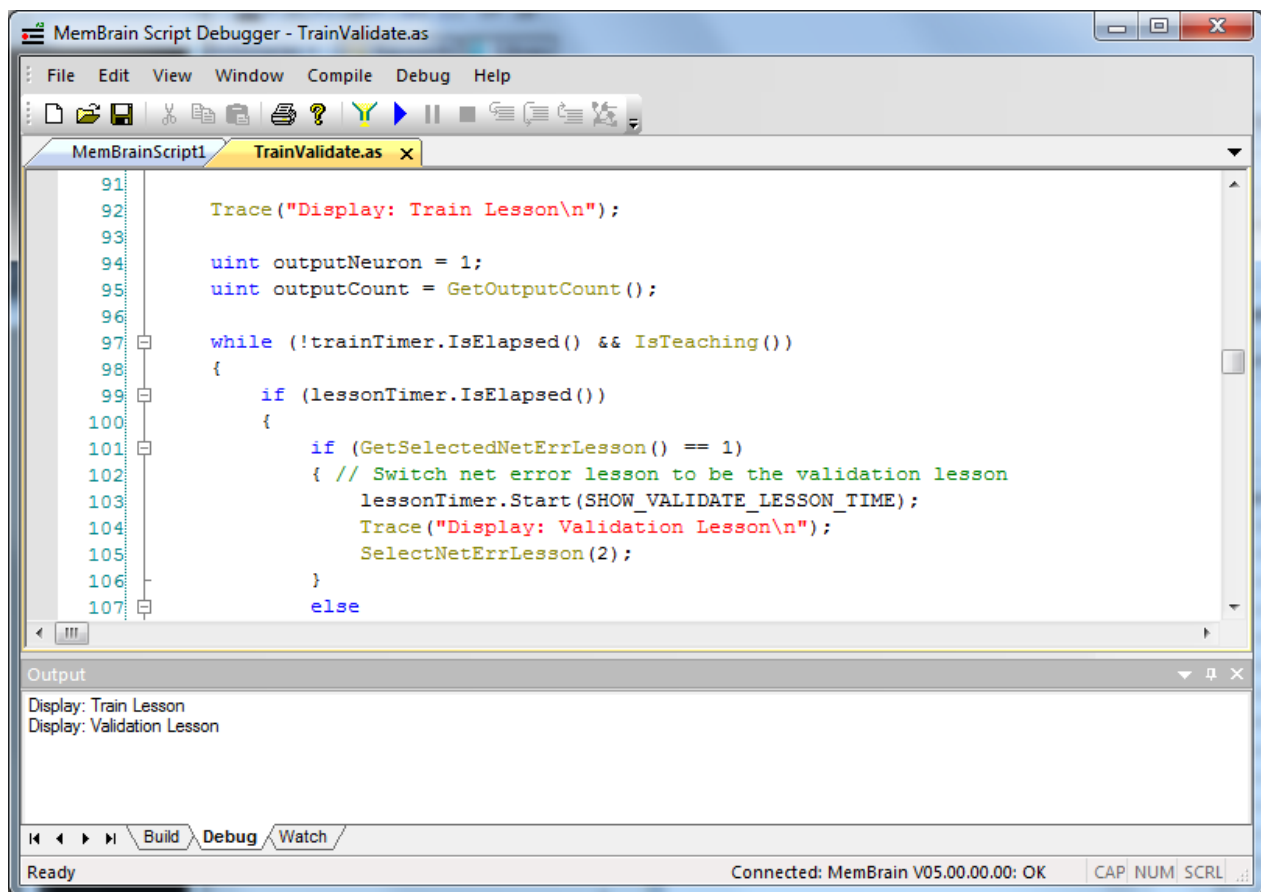
The debugger will automatically switch to the Build Output window, start the script compilation and - if no errors occur during compilation - start the script within MemBrain. The debugger will automatically halt script execution at the first instruction after the program entry point as shown in the following picture.



To let the script run repeat the Run/Continue action as above.

The Debug output window

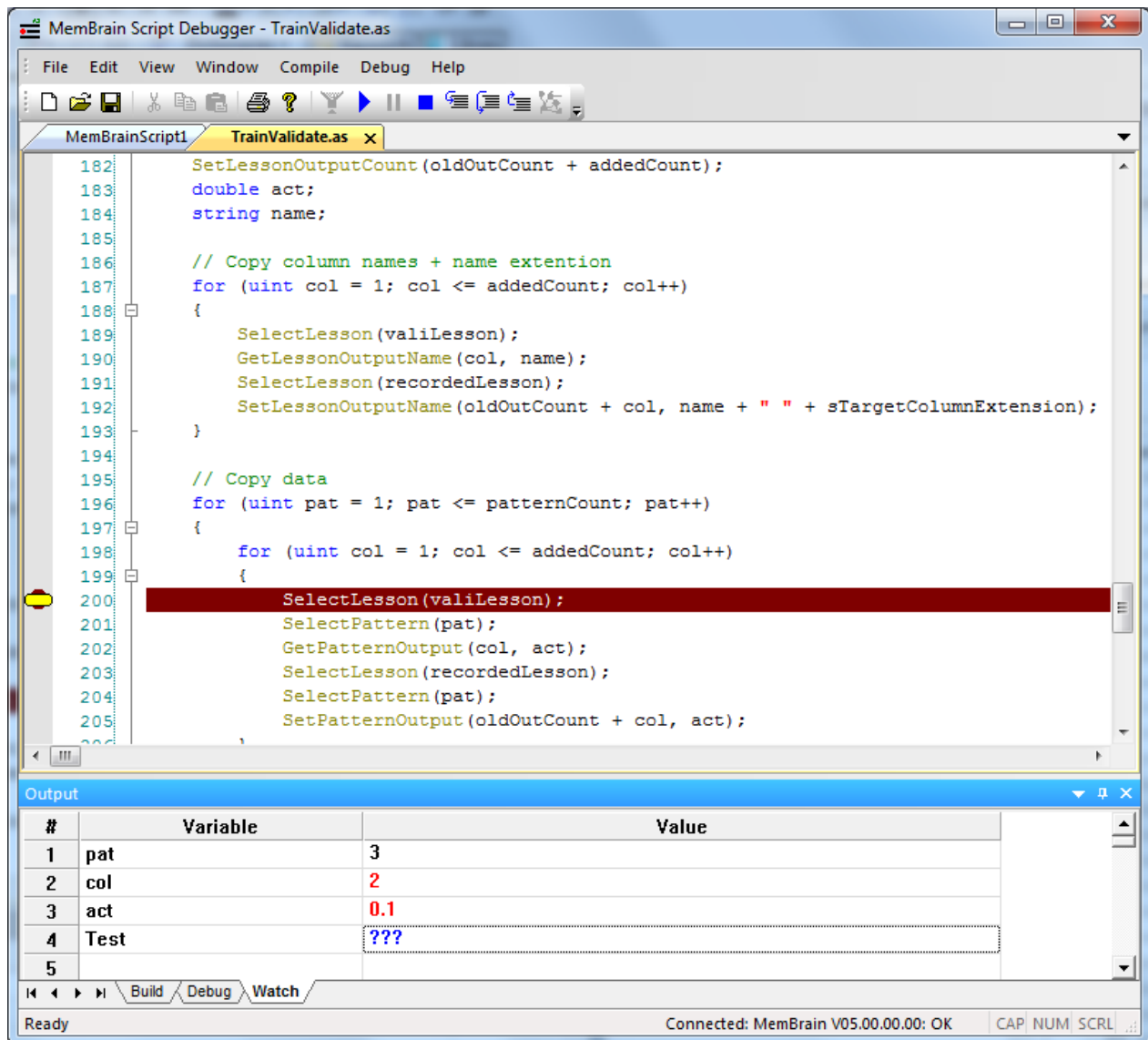
The debug output window pane will capture and display all trace messages emitted by the script:



The text messages printed to the Debug window pane in this example have been emitted by the Trace instructions in the lines 92 and 104 of the shown script, respectively.

The Watch window

Values of script variables can be observed in the watch window whenever the script execution is halted in the debugger. The following example shows a watch window with three observed variables.






Note the coloring of the variable values: Values displayed in red have changed since the script had been halted the last time. Black values indicate unchanged values. The variable 'Test' is not known or at least not visible in the current scope and thus is indicated by three blue question marks.

Use the menu command <Debug><Clear Watch Window> to remove all expressions from the watch list.



Step In, Step Over, Step Out

You can perform step in, step out or step over operations with the debugger using the following options:

- Toolbar icons , , 
- Keyboard shortcuts F11, F10, SHIFT + F11
- Corresponding menu commands in the menu section <Debug>

Pause or Stop script execution

Script execution can be paused or stopped (i.e. aborted) by use of the following.

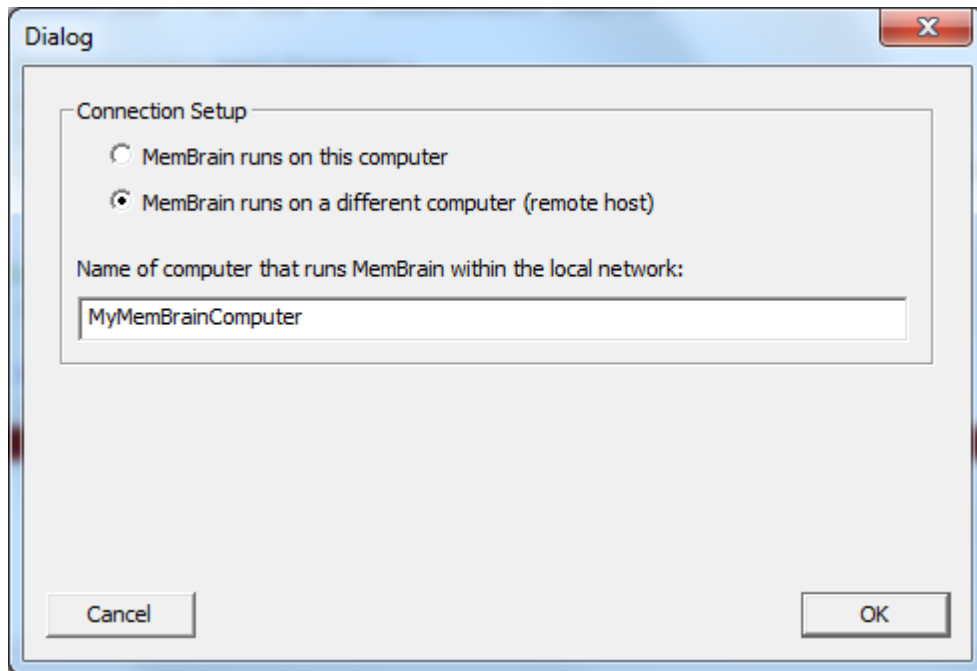
- Toolbar buttons  and 
- Corresponding menu commands or shortcuts as listed in the menu section <Debug>

Block Indentation

Selected text blocks can be indented or unindented using <TAB> or <SHIFT> + <TAB>, respectively.

Set up debugger connection

The MemBrain Script Debugger is able to run on the same machine as MemBrain or even on a different machine within the same LAN (Local Area Network). The debugger is pre-configured after installation to run on the same machine as MemBrain. However, in case you want to run the debugger on a different machine you can set up the debugger connection accordingly: Start the debugger manually on the desired machine and use the menu option <Debug><Setup Debugger Connection...> which will bring up the following dialog:



Select the second option and enter the name of the machine within the LAN where MemBrain runs. If the remote machine exists and if MemBrain is running on this remote machine the debugger will automatically connect to the MemBrain instance on the remote machine. You can identify a successful connection in the status line of the debugger.

The MemBrain Script Debugger features other options beyond those listed. Where these are not self-explanatory or in case you should have further questions or observations with respect to the debugger don't hesitate to contact Thomas Jetter directly via the contact information provided [here](#).

Source Code Generation

MemBrain is able to export trained nets into source code so that they can be incorporated into own software running on arbitrary targets including micro controllers.

Currently the following source code export formats are available.

[C-Code Generation](#)

C Code

C Code Generation

MemBrain can generate C code of the current neural net. You can incorporate the generated source code into own software using any C compiler.

This allows to run trained MemBrain neural nets on other targets than Windows, e.g. on micro controller

platforms. The generated C code is limited to only one neural net at a time. If you want to use multiple and possibly different neural nets in parallel in your custom application please use the C++ code generation instead.

How it works:

[Configure the source code generation feature](#) (first time only)

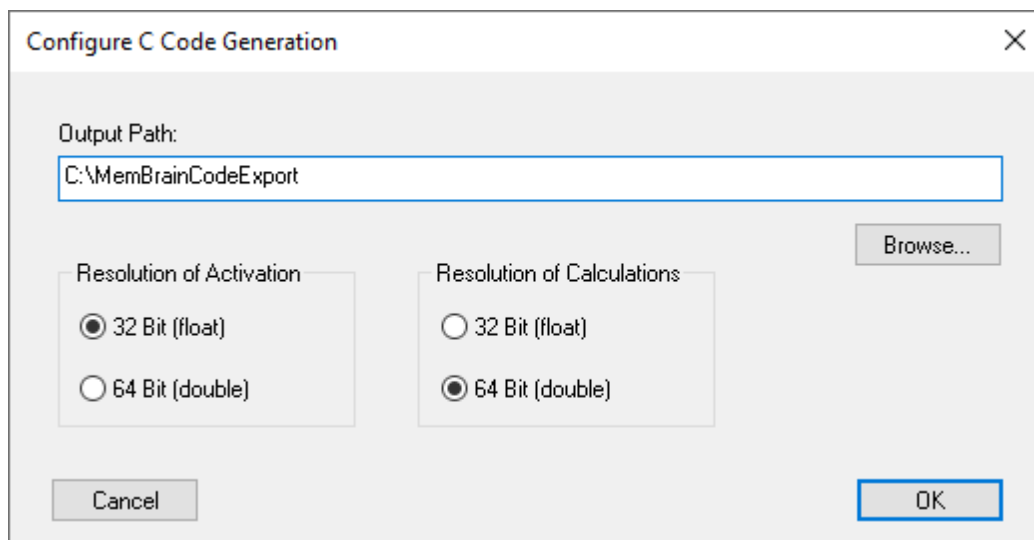
[Generate the source code](#)

[Build your application using the generated source code](#)

Configuration

The MemBrain C source code generation can be configured by selecting **<Code-Generation><C Code><Configuration...>** from the MemBrain menu.

The following dialog will appear:



You can specify an output path for the generated files as well as the resolution used for

- the activation and corresponding normalization parameters of the neurons.
- the calculations performed by the C code library together with all other neural net parameters

Note:

When running the generated code on a micro controller target then the 32 Bit option should be considered since for most applications the 32 Bit accuracy is sufficient and it requires significantly less memory and performance resources compared to the 64 Bit version.

Generate Code

To generate C-Source Code from the currently loaded net execute the command **<Code Generation><C Code><Generate Code>**.

You will be prompted about the successfully completed code generation.

The generated code consists of the following two files.

NeuralNetDef.h
NeuralNetDef.c

which are generated in the directory adjusted as part of the [C Code Generation Configuration](#).

These two files together with the *MemBrain C-Code Generation Library* form the code that has to be incorporated into your own [software build process](#).

The code of the *MemBrain C-Code Generation Library* is configured by the MemBrain exported header file `NeuralNetDef.h` through pre-processor directives in an optimal way with respect to needed resources. According to the features you use in your net only the portions of the library that are really needed to implement your net are invoked for the compiler. Thus, a simple, time invariant feed forward net does not incorporate the same code as a more advanced, time variant net does.

You do not have to configure any of these optimizations. MemBrain automatically takes care of this during the code generation procedure on basis of its internal advanced net analysis. Thus, it is guaranteed that your compiled C code always represents an optimal trade-off between required features and performance.

Build Your Application

The C code for any neural net always consists of the two files

NeuralNetDef.h
NeuralNetDef.c

that have been generated by MemBrain for a certain neural net together with the *MemBrain C Code Generation Library* that consists of the following files that are **located in the sub directory 'C_CODE'** of the MemBrain installation directory.

NN_Types.h
NeuralNet.c
NeuralNet.h
Neuron.c
Neuron.h
NeuralLink.c
NeuralLink.h
Random.c
Random.h

It is strongly recommended to make a copy of these files and only use the copies to build your application since this reduces the risk of unintentionally editing the files.

Depending on your compiler you may need to **edit** the file **NN_Types.h** to align the data types required by the MemBrain C-Code Generation Library to your target.

Add all the .c files to your application build environment, include the header file 'NeuralNet.h' in your application and use the interface functions listed in the header file **NeuralNet.h** to access the neural net:

/// Initialize and reset the neural net. **Call this function at least once in first place!!**

void NeuralNetInitAndReset(void);

/// Let the net perform one think step

void NeuralNetThinkStep(void);

/// Apply an activation value to an input neuron. <inputIdx> can range from 0 to MB_IN_NEURON_COUNT - 1

BOOL NeuralNetApplyInputAct(T_NEURON_COUNT inputIdx, T_ACT_FLOAT act);

/// Get the activation value of an output neuron. <outputIdx> can range from 0 to MB_OUT_NEURON_COUNT - 1

BOOL NeuralNetGetOutputAct(T_NEURON_COUNT outputIdx, T_ACT_FLOAT* pAct);

/// Get the output value of an output neuron. <outputIdx> can range from 0 to MB_OUT_NEURON_COUNT - 1

BOOL NeuralNetGetOutputOut(T_NEURON_COUNT outputIdx, FLOAT32* pOut);

Note: Most of the times the output value of an output neuron is of no interest but the activation value is. The activation value is what you train against and it is always in units according to the user normalization ranges.

The output always is normalized to internal ranges and may be subject to recovery times and random firing according to the selected neuron parameters.

C++ Code

C++ Code Generation

MemBrain can generate C++ code of the current neural net. You can incorporate the generated source code into own software using any C++ compiler.

This allows to run trained MemBrain neural nets on other targets than Windows, e.g. on micro controller platforms. The C++ code generation - as opposed to the C code generation - allows to generate C++ code for multiple different neural nets and use them in parallel within the same custom C++ user application.

How it works:

[Configure the source code generation feature](#) (first time only)

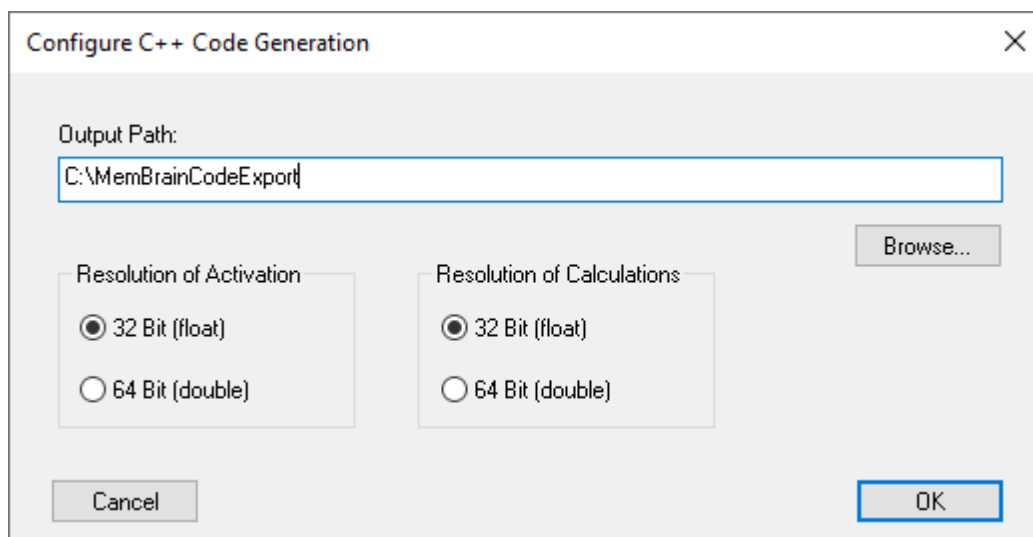
[Generate the source code](#)

[Build your application using the generated source code](#)

Configuration

The MemBrain C++ source code generation can be configured by selecting **<Code Generation><C++ Code><Configuration...>** from the MemBrain menu.

The following dialog will appear:



You can specify an output path for the generated files as well as the resolution used for

- the activation and corresponding normalization parameters of the neurons.
- the calculations performed by the C++ code library together with all other neural net parameters

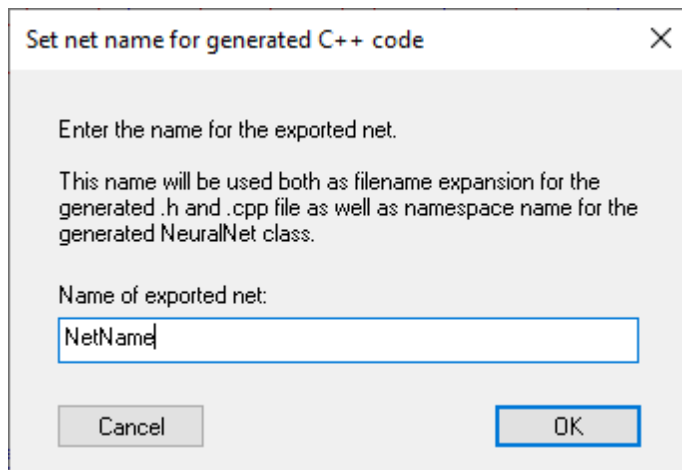
Note:

When running the generated code on a micro controller target then the 32 Bit option should be considered since for most applications the 32 Bit accuracy is sufficient and it requires significantly less memory and performance resources compared to the 64 Bit version.

Generate Code

To generate C++ source code from the currently loaded net execute the command **<Code Generation><C++ Code><Generate Code...>**.

MemBrain will show the following dialog that lets you enter the name of the exported net.



MemBrain will use this string for both file name generation and for definition of a net specific namespace in the generated code.

The dialog is prefilled with a string that is derived from the file name of your currently loaded net. The displayed string will only consist of characters that are allowed to be part of an identifier in C++ language. You can edit the string for the net name but you will only be allowed to use such valid characters. All invalid characters will automatically be replaced by underscores '_' during editing.

Click <OK>. You will be prompted about the successfully completed code generation.

The generated code consists of the following two files.

NeuralNet_*NetName*.h
NeuralNet_*NetName*.cpp

which are generated in the directory adjusted as part of the [C++ Code Generation Configuration](#). *NetName* is the name for the net you entered in the dialog above.

These two files together with the *MemBrain C++ Code Generation Library* form the code that has to be incorporated into your own [software build process](#).

The code of the *MemBrain C-Code Generation Library* is configured by the MemBrain exported header file *NeuralNet_*NetName*.h* through pre-processor directives in an optimal way with respect to needed resources.

According to the features you use in your net only the portions of the library that are really needed to implement your net are invoked for the compiler. Thus, a simple, time invariant feed forward net does not incorporate the same code as a more advanced, time variant net does.

You do not have to configure any of these optimizations. MemBrain automatically takes care of this during the code generation procedure on basis of its internal advanced net analysis. Thus, it is guaranteed that

your compiled C++ code always represents an optimal trade-off between required features and performance.

Build Your Application

The C++ code for any neural net always consists of the two files

NeuralNet_NetName.h
NeuralNet_NetName.cpp

that have been generated by MemBrain for a certain neural net together with the *MemBrain C++ Code Generation Library* that consists of the following files that are **located in the sub directory 'CPP_CODE'** of the MemBrain installation directory.

MB_NeuralNet.h
MB_NeuralNet.hpp
MB_Neuron.h
MB_Neuron.hpp
MB_Random.h
MB_Random.hpp
MB_NeuralLink.h
MB_NeuralLink.hpp
MB_NN_Types.h

Note: **Do not include any of the above library files in your user code.** All files are automatically included as required by the generated files for your net. Instead, copy the above files to your C++ source code repository and ensure that the include path for your compiler is set accordingly in order to be able to find these files during the build process.

Depending on your compiler you may need to **edit** the file **NN_Types.h** to align the data types required by the MemBrain C++ Code Generation Library to your target.

Add the **NeuralNet_NetName.cpp** file to your application build process and include the header file '**NeuralNet_NetName.h**' in your application. Create new net instances and use their interface functions listed in the header file **MB_NeuralNet.h** to access the neural net.

```
public:
    /// Constructor
    NeuralNet();

    /// Activate this instance of the NeuralNet. Must be called whenever the active
    instance shall change. Not required when only one instance exists.
    void ActivateInstance();

    /// Initialize and reset the neural net. Call this function at least once in first
    place!!
    void InitAndReset(void);

    /// Let the net perform one think step
    void ThinkStep(void);

    /// Apply an activation value to an input neuron. <inputIdx> can range from 0 to
    IN_NEURON_COUNT - 1
    bool ApplyInputAct(T_NEURON_COUNT inputIdx, T_ACT_FLOAT const act);

    /// Get the activation value of an output neuron. <outputIdx> can range from 0 to
    OUT_NEURON_COUNT - 1
    bool GetOutputAct(T_NEURON_COUNT outputIdx, T_ACT_FLOAT& act) const;

    /// Get the output value of an output neuron. <outputIdx> can range from 0 to
    OUT_NEURON_COUNT - 1
    bool GetOutputOut(T_NEURON_COUNT outputIdx, MB_FLOAT& out) const;
```

Always use *NetName* as namespace when accessing properties of your net.

Example:

```
#include "NeuralNet_NetName.h"           // File generated by MemBrain via net
export name "NetName"
#include "NeuralNet_OtherNetName.h"      // File generated by MemBrain via net
export name "OtherNetName"

// We use three nets in this example
NetName::NeuralNet myNet1;              // Net 1 of type "NetName"
NetName::NeuralNet myNet2;              // Net 2 of type "NetName"
OtherNetName::NeuralNet myNet3;         // Net 3 of type "OtherNetName"

// Initialize and reset the nets
myNet1.InitAndReset();
myNet2.InitAndReset();
myNet3.InitAndReset();

DoSomethingOnNet(myNet1);
DoSomethingOnNet(myNet2);
DoSomethingOnNet(myNet3);
```

With the following overloaded example functions. Note that the functions basically have the same implementations but are using different namespaces according to the net types they are designed for.

```
// This function does something with a net of type <NetName::NeuralNet>
void DoSomethinkOnNet(NetName::NeuralNet& rNet)
{
    using namespace NetName;
    std::array<T_ACT_FLOAT, OUT_NEURON_COUNT> outAct{ 0 };
    int i;
    T_NEURON_COUNT idx;
    T_ACT_FLOAT actIn{ (T_ACT_FLOAT)0 };
    T_ACT_FLOAT increment{ (T_ACT_FLOAT)0.05 };

    // myNet1 and myNet2 are of the same type. In this case we have to call this
    // method to set the operation focus to the current instance
    rNet.ActivateInstance();

    for (idx = 0; idx < IN_NEURON_COUNT; idx++)
    {
        rNet.ApplyInputAct(idx, actIn);
        actIn += increment;
    }

    for (i = 0; i < 1; i++)
    {
        rNet.ThinkStep();
        for (idx = 0; idx < OUT_NEURON_COUNT; idx++)
        {
            rNet.GetOutputAct(idx, outAct[idx]);
            TRACE("Output %d = %g\r\n", idx, outAct[idx]);
        }
        TRACE("\r\n");
    }
}

// This function does something with a net of type <OtherNetName::NeuralNet>
static void DoSomethinkOnNet(OtherNetName::NeuralNet& rNet)
{
    using namespace OtherNetName;
```

```

std::array<T_ACT_FLOAT, OUT_NEURON_COUNT> outAct{ 0 };
int i;
T_NEURON_COUNT idx;
T_ACT_FLOAT actIn{ (T_ACT_FLOAT)0 };
T_ACT_FLOAT increment{ (T_ACT_FLOAT)0.05 };

// There is only one instance of OtherNet::NeuralNet. We don't need to call
// ActivateInstance(). It does not hurt to do it, however.
rNet.ActivateInstance();

for (idx = 0; idx < IN_NEURON_COUNT; idx++)
{
    rNet.ApplyInputAct(idx, actIn);
    actIn += increment;
}

for (i = 0; i < 1; i++)
{
    rNet.ThinkStep();
    for (idx = 0; idx < OUT_NEURON_COUNT; idx++)
    {
        rNet.GetOutputAct(idx, outAct[idx]);
        TRACE("Output %d = %g\r\n", idx, outAct[idx]);
    }
    TRACE("\r\n");
}
}

```

Note: Most of the times the output value of an output neuron is of no interest but the activation value is. The activation value is what you train against and it is always in units according to the user normalization ranges.

The output always is normalized to internal ranges and may be subject to recovery times and random firing according to the selected neuron parameters.

The MemBrain DLL

Many functionalities of MemBrain are also available as a DLL (Dynamic Link Library) version.

With the MemBrain DLL you can incorporate neural nets that have been designed with MemBrain into your own applications no matter which programming language you use. The DLL features:

- Loading an arbitrary number of neural nets from MemBrain files
- Assigning values to the input neurons of the nets
- Performing simulation steps of the nets
- Reading back the results from the output neurons of the nets
- Saving the nets back to file
- All functionalities to train the nets, to create, load, import and export data sets and much more
- All functionalities to create and edit neural nets

By this means it is easily possible to incorporate neural nets from MemBrain into a production system without the need of having MemBrain running in parallel and without having to deal with the TCP/IP interface or the scripting language of MemBrain. Additionally, applications can be designed that create and/or edit neural nets during run time and thus can dynamically adapt to new problems.

The MemBrain DLL is included in the installation packet of MemBrain, the corresponding files will be copied to a separate sub folder 'DLL' automatically during the installation. In the download area of the MemBrain homepage you will find an example on how to use the DLL from within Microsoft Excel VBA (Visual Basic for Applications). Java users are provided with a JNI wrapper DLL and a corresponding Java class to directly access the DLL from Java programs. The Java class file contains comments on how to use the files. Both, the wrapper DLL and the Java class file are copied to the DLL folder during intallation automatically.

An object oriented set of C# wrapper classes is available for free, too (see MemBrain homepage download area). Certainly the MemBrain DLL also is accessible from the C/C++ programming language. Detailed information on the DLL is included in the corresponding header file "MemBrain_inc.h".

The MemBrain DLL also supports reading and saving [encrypted](#) neural nets and lessons.

Encryption

MemBrain supports strong encryption of files for neural nets and lessons using password protection (AES 256 bit encryption).

You should use encryption if you don't want that your neural nets or MemBrain lessons can be opened by other persons without the correct password. Encryption is supported by both the MemBrain main application and also by the MemBrain dll. Thus, it is also possible to distribute your own neural net files in a protected way in order to use it with the [MemBrain DLL](#) in your own application.

Note: Be sure to remember your passwords! If you forget the password to a file which has been encrypted by MemBrain It is virtually impossible to open the file again. 256 bit AES encryption is one of the strongest encryption algorithms available today!

The following sections explain how encryption and passwords are handled in MemBrain.

[Set/Remove passwords for nets and lessons](#)

[Open encrypted files](#)

[Use default password for opening files](#)

Set/Remove Passwords

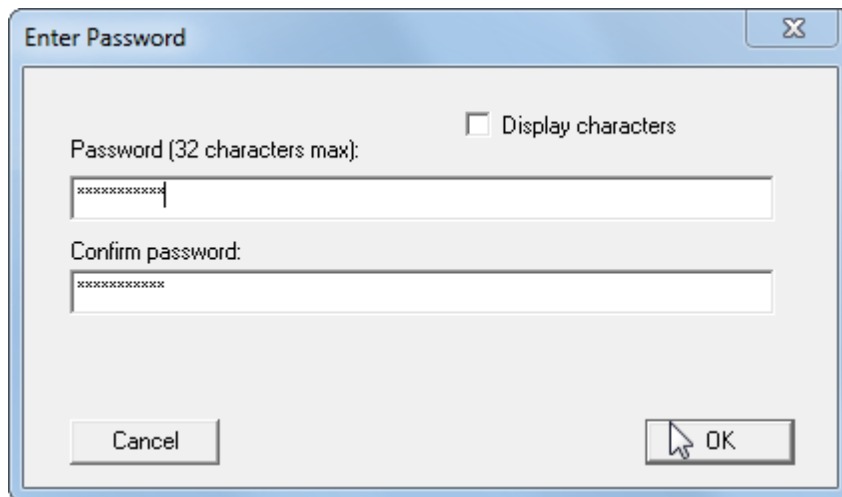
To save neural nets or lessons in encrypted format select one of the following:

- a) Select <File><Set File Password (Encryption)...> for assigning a password to the currently open net. Encryption will become effective when you save the net to file.

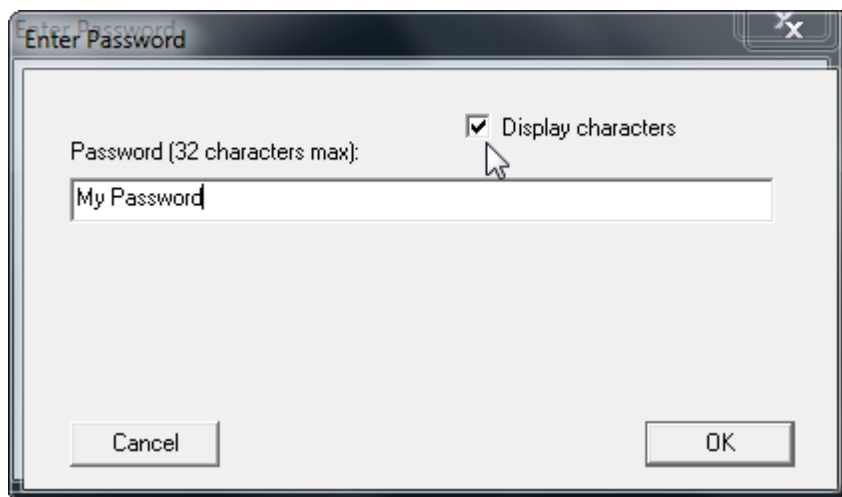
OR

- b) In the Lesson Editor menu select <Lesson Files><Set Lesson Password (Encryption)...>. Again, Encryption will become effective when you save the lesson to file. Note that this action sets the password and enables encryption only for the currently selected lesson. Different passwords can be assigned to different lessons by repeating this action for the desired lessons.

The following dialog will open:



You have to enter the password twice for confirmation. Alternatively, you can check the box 'Display characters' which will show the password in readable form and also suppress the second confirmation edit field:



Note: Only select this box if you are sure that nobody else than yourself can see your screen! The box will be disabled by default whenever the dialog opens in order to protect your passwords from being read by others. For the same reason, when the check box is disabled then it is also not possible to copy text from the edit fields into the clip board.

You can always assign a new password to an open file without entering the current password. Also, it is possible to remove the password protection from the currently open net or currently edited lesson by selecting:

- a) Select <File><Remove File Password (Encryption)...> for removing the password protection from the currently open net. Non-encrypted storage will become effective when you save the net to file.

OR

- b) In the Lesson Editor menu select <Lesson Files><Remove Lesson Password (Encryption)...>. Again, non-encrypted storage will become effective when you save the lesson to file. Note that this action removes the password and disables encryption only for the currently selected lesson.

Open encrypted files

MemBrain automatically detects when an encrypted file shall be opened and will ask you for entering the correct password in case the currently set [default password](#) does not work in combination with the file to be opened.

The following dialog will open:

Enter the password and click <OK>. If you want the password to be displayed in readable form during entering then check the box 'Display characters':

Note: Only select this box if you are sure that nobody else than yourself can see your screen! The box will be disabled by default whenever the dialog opens in order to protect your passwords from being read by others. For the same reason, when the check box is disabled then it is also not possible to copy text from the edit fields into the clip board.

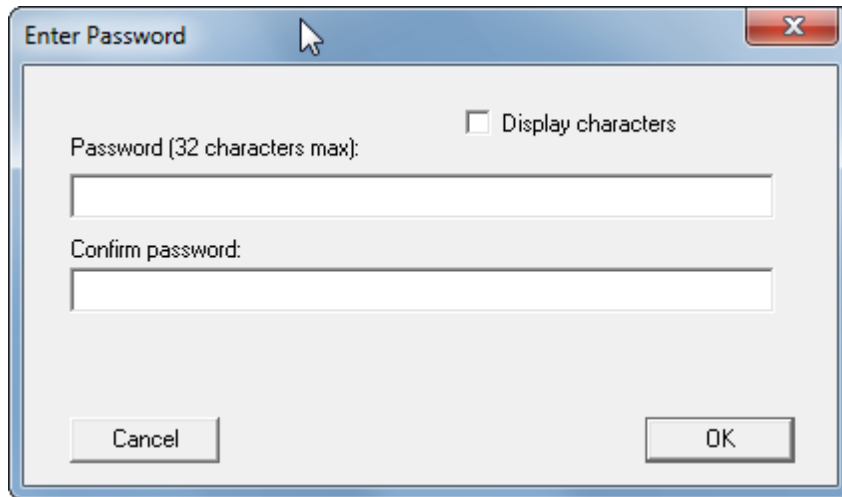
If you enter the correct password the file will be opened as usual in MemBrain. Note that you then also can [remove the password from the file](#) and store it in an unencrypted format if you like.

By activating the check box 'Set as default...' in the dialog the password will be set as the [default password](#) for this MemBrain session in case the open action succeeds.

Use Default Password

If you frequently work with encrypted files and if all these files use the same password then it is most convenient to enter this password as the default password every time MemBrain has been started. When an encrypted file is to be opened MemBrain will always try to open the file with the default password first. Only if this fails MemBrain will ask for another password to be entered for opening the file.

In order to set the default password in MemBrain select <File><Set Default Password (Encryption)...>. The following dialog appears:



Enter your default password for this MemBrain session and click OK. You can also check the box 'Display characters' so that your entered password is displayed in readable form. The second password confirmation box will not be shown in this case.

Note: Only select this box if you are sure that nobody else than yourself can see your screen! The box will be disabled by default whenever the dialog opens in order to protect your passwords from being read by others. For the same reason, when the check box is disabled then it is also not possible to copy text from the edit fields into the clip board.

Note: The default password is not stored persistently by MemBrain. I.e., the password is cleared automatically each time you exit MemBrain in order to protect your data against unwanted access by others.

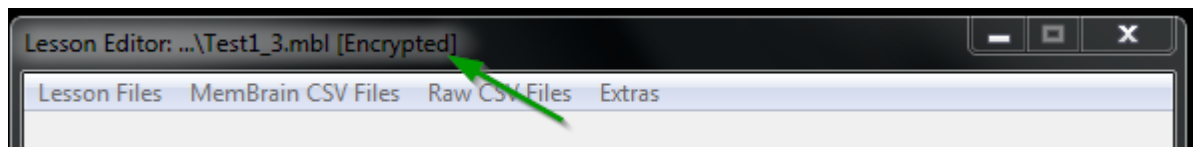
Check Encryption Status of Files

If the neural net currently open in MemBrain is encrypted then MemBrain displays this in its main window header by expanding the name of the open file with [Encrypted]:



Note that after setting a password the file is already displayed as 'Encrypted'. However, setting a password will not automatically save the file to disk. I.e. the file may still be stored in an unencrypted format on the disk. So be sure to save a file after enabling the encryption for this file.

Similarly, the Lesson Editor displays in its window header if the currently edited lesson is encrypted:



Note that if you open an encrypted file it will stay encrypted with the same password unless you [set another password or remove the password](#). This is also the case if you save the file to a different location or file name.

Multi Core Support / Multi Threading

MemBrain can optimize processing of neural nets by distributing calculation to several cores (CPUs) of a multi core computer.

In principal, there are two separate ways to perform multithreading on a neural net:

1. Lesson-Based Multithreading during teaching

The lesson used for training of the neural net is split into multiple parts, each single part is then dedicated to its own Thread/CPU/Core. This is the most effective and thus most favourable way of using multi threading. The [Teacher Manager](#) indicates for each training algorithm in MemBrain whether the corresponding teacher is able to support this kind of multi threading or not.

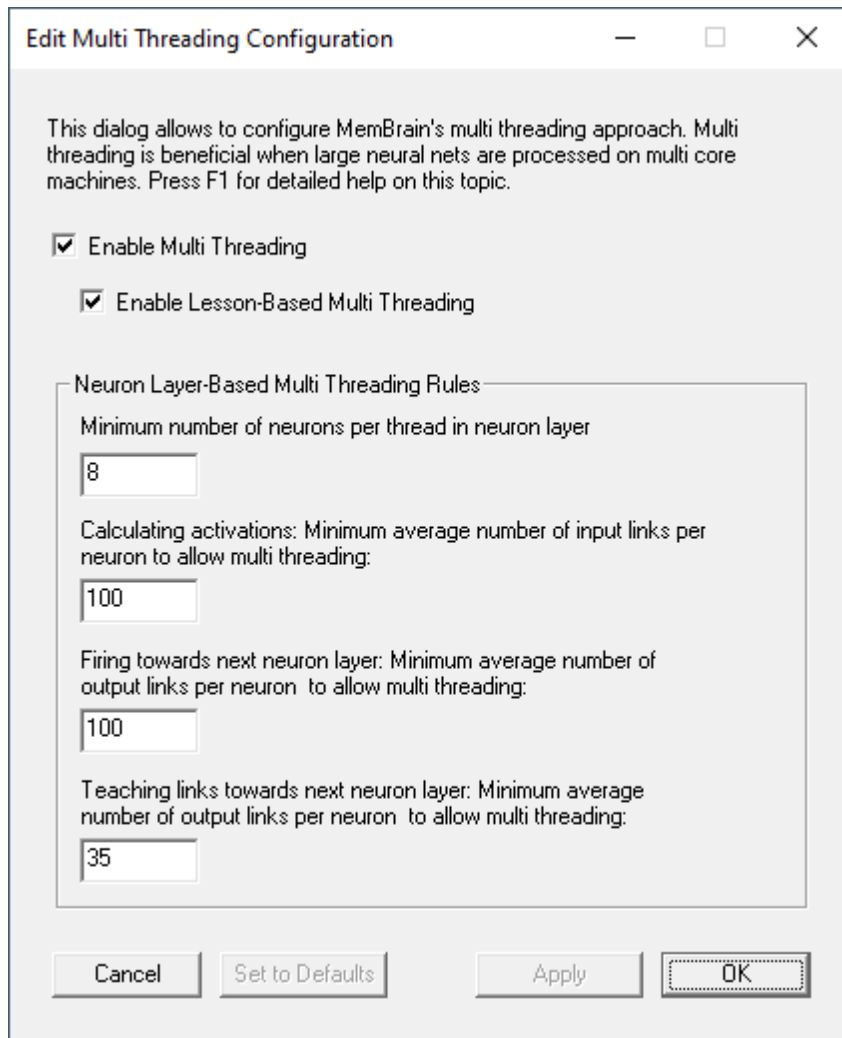
See [here](#) for more details on Lesson-Based Multithreading in MemBrain.

2. Layer-Based Multithreading during teaching and thinking

Each neuron layer in a neural net is split into multiple groups of neurons and each neuron group is assigned to its own Thread/CPU/Core. Doing so implies several restrictions and a quite complex set of optimization parameters.

See [here](#) for more details on Lesson Based Multithreading in MemBrain.

In order to review or edit MemBrain's multi threading parametrization select <Edit><Configure Multi Threading>. The following dialog appears:



Besides editing the single parameters which configure the layer-based multi threading you can also disable the multi threading in general if you want to. This way you can at any time test if your current multi threading configuration speeds up things or may even make things worse compared to the single threading approach.

Note that the dialog is non-modal. I.e. you can open this dialog even during training of MemBrain and change the multi threading approach on-the-fly by either clicking <Apply> (leaves dialog open) or <OK> (applies changes and closes the dialog).

If you should find parameter values that improve MemBrain's performance compared to the default values then feel free to report them to Thomas Jetter (see [Contact](#) information) via E-Mail, thanks for this in advance!

Lesson Based Multithreading

Lesson-based multi threading

Lesson-based multi threading is only available for the training/teaching process. Certainly, the training of a neural net is where most of the processing time of a computer is required anyway.

Splitting the current lesson into parts for the training procedure actually requires each thread or teacher instance to work with its own copy of the neural net and then to merge the training results back into one resulting neural net after each lesson run of the teacher. This is required to allow each teacher instance / thread to operate independently from and in parallel to the other instances / threads.

Thus, when lesson-based multi threading is started, a certain preparation time is needed by MemBrain before the training can actually start. The length of this time span mainly depends on the size and complexity of the net. MemBrain displays the message 'Preparing Multithreading...' in its status bar during this time.

Other side effects of lesson-based multi threading are:

- Possible problems with time-variant nets

Since the lesson has to be split into parts for the training (e.g. 4 partial lessons on a quad core work station), the time wise / pattern-order wise dependencies between these partial lessons cannot be kept during training since the partial lessons are used for the training in parallel (which is the actual goal of the lesson-based multi threading).

I.e., each partial lesson within itself is trained in the correct order of the patterns. The transitional dependencies between the patterns in the end of a partial lesson and the patterns in the beginning of the next partial lesson cannot be kept, however.

MemBrain issues a warning in the beginning of the training in case the lesson-based multi threading may cause conflicts with the time variance/pattern-order dependencies.
- Inability to dynamically rename the output winner neuron during training

Since in the background MemBrain needs to work with multiple neural net copies during lesson-based multi threaded training (e.g. 4 copies of the net on a quad core work station), the teacher cannot rename the winner neurons in the net according to the currently trained lesson: After a lesson run has been completed it would not be possible to determine the correct re-naming activities when merging the training results.

MemBrain issues a warning in the beginning of the training in case the lesson-based multi threading may cause conflicts with the winner renaming feature. The [Lesson Editor](#) allows to determine the final winner neuron names manually after the training has been concluded.

Layer Based Multithreading

Layer-based multi threading

With this kind of multi threading, MemBrain has to respect certain rules in order to not invalidate the overall outcome of the calculations: The order of calculations in a neural net is subject to certain restrictions. The most prominent of these restrictions is the correct order of calculating layers in a neural net: The neuron activations of a neuron layer may only be calculated if all calculations of the previous neuron layer(s) have been completed. This means that - when using multi threading - MemBrain may not assign different neuron layers to different cores of the computer: Since each layer needs to wait for the outcomes of its input layer(s) before it may perform its own calculations, such a distribution would not make sense. Multithreading only makes sense if the different cores are allowed to actually perform their work in parallel.

As a result of the above said, MemBrain has to split each single layer in the neural net into different portions and assign these portions of the same single layer to the different cores of the machine for calculations.

Moreover, there is significant overhead associated to multithreading due to the need to synchronize

calculations between different threads. This means that layer-based multithreading only comes beneficial if the size and complexity of the neural net grows beyond certain thresholds. These thresholds can be configured by the user. However, the default values have been already chosen so that on most machines and for most net architectures the configuration should already be optimal.

You may want to play with the settings and see if you are able to get more performance out of your machine on certain high complexity neural nets. In case you should feel that you have become lost with the settings and you might have made things worse instead of improving them you can reset MemBrain's multi threading parameters to their default values, too.

In order to review or edit MemBrain's multi threading parametrization select <Edit><Configure Multi Threading>. The following dialog appears:

Edit Multi Threading Configuration

This dialog allows to configure MemBrain's multi threading approach. Multi threading is beneficial when large neural nets are processed on multi core machines. Press F1 for detailed help on this topic.

☒ Enable Multi Threading

☒ Enable Lesson-Based Multi Threading

Neuron Layer-Based Multi Threading Rules

Minimum number of neurons per thread in neuron layer
8

Calculating activations: Minimum average number of input links per neuron to allow multi threading:
100

Firing towards next neuron layer: Minimum average number of output links per neuron to allow multi threading:
100

Teaching links towards next neuron layer: Minimum average number of output links per neuron to allow multi threading:
35

Cancel Set to Defaults Apply OK

The majority of the settings therein refer to layer-based multi threading only and are irrelevant if Lesson-based multi threading is used.

TCP Weblink

Since version 02.01.00.10 MemBrain supports connections to other computers and/or I/O devices over one or more TCP connections.

Through this the following features are achieved:

- Neural nets can be spread over an arbitrary number of MemBrain instances running on several computers or even on one computer only.

- Other applications can use a common interface to communicate with MemBrain so that virtually any I/O hardware can be connected to MemBrain by implementing a 'driver' (actually a service) that supports MemBrain's TCP based protocol.

This chapter and its sub chapters explain how different MemBrain instances can be connected to each other in order to build larger nets spread over different computers.

Also details on MemBrain's communication protocol are provided to enable people that want to connect their hard- and software to MemBrain to implement appropriate communication software.

Operation

This chapter provides you with all information needed to use MemBrain's Weblink to build nets spread over several MemBrain instances and/or hardware I/O drivers.

It does not go into details about the used communication protocol and sequences. If you want to write your own "driver" to enable MemBrain to access other software or hardware you will need to go into details using the chapter [Protocol](#).

Nevertheless you should first read through the chapter <Operation>

Public and Extern Neurons

All TCP based connections of MemBrain to other software or other instances of MemBrain are realized through a concept of so called **Public** and **Extern** neurons:

The idea behind this is that neurons that are members of a certain neural net are made **public** so that they can be accessed from other neural nets controlled by other software applications.

To do this the public neurons of a net can be **invoked** as **Extern** neurons in other nets and there treated just like any other neuron. Nevertheless it is important to mention that even if a public neuron is invoked as extern neuron in one or more other nets it still does exist only once i.e. inside the MemBrain instance where it is made public. All calculations and output signal generation with respect to that neuron are performed on the original neuron that has been made public. Extern neurons in other nets are just some kind of images or access points to the original public counterparts behind them.

As a consequence of the things said above extern neurons cannot be edited with respect to their operational parameters like the activation function or thresholds for example. However it is possible to edit some properties of the external neurons e.g. their name inside the net where they are invoked or also their functionality in the net (output/input/hidden).

Public neurons in MemBrain are identified by the letter 'P' in their upper right corner:



Extern neurons have an 'E' displayed instead:



As a general rule, **one public** neuron can be invoked into several other nets thus having **more than one**

extern counterpart. On the opposite an **extern** neuron always has **only one public** counterpart to which it provides access.

Access to a public neuron via an extern counterpart is implemented bidirectional which means that the activation and the output signal of the public neuron are transmitted to the extern neuron and the input signals arriving at the extern neuron are transmitted to the public neuron where they are interpreted as additional input.

Activate/Deactivate Weblink

There are two menu items that enable or disable the Weblink of a MemBrain instance in general:

<Weblink><Activate Weblink>

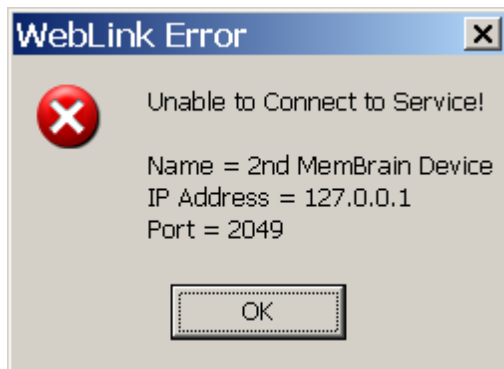
and

<Weblink><Deactivate Weblink>

Note that the latter of the two breaks all eventually active connections to and from other devices without prior request!

Only one of the both menu items is available at a time depending on the fact if the Weblink is currently active or not. Note that an active Weblink does not mean that there are already connections established to other devices.

Nevertheless if there are extern neurons in the net then MemBrain will automatically try to connect to the corresponding public counterpart neurons when the Weblink is activated. If it is not possible to connect to the corresponding IP address(es) and port(s) you will get a message e.g. like this:

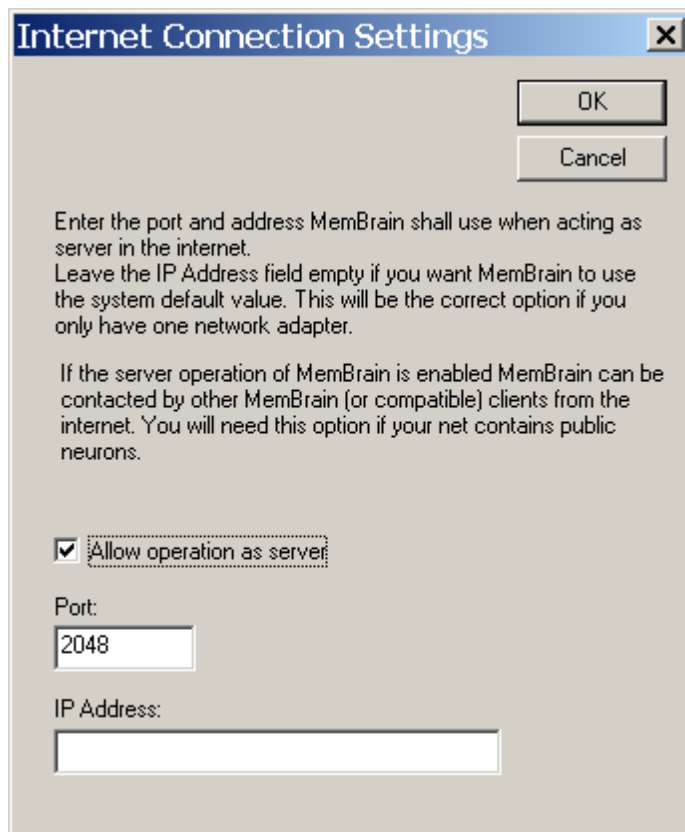


Note that if you do not get a message this doesn't mean that all of the extern neurons could have been successfully connected to their public counterparts. One reason for this can be that the corresponding public neuron does not exist on the connected service (e.g. other MemBrain instance). Check for the extern neuron's [Connection State](#) to ensure that they are properly connected.

TCP Connection Logic

As a general rule for all devices that want to communicate using the MemBrain protocol the TCP connection must be requested by the device that wants to invoke public neurons located on another device. As a consequence of this every device that publishes neurons has to act as connection server that is, listen on a certain IP address and port for other MemBrain devices that want to connect.

The settings used for listening to TCP connection requests can be edited by the menu option <WebLink><Server Settings...> which will open the following dialog.



You can enter the port this MemBrain instance shall be awaiting communication requests on as well as an IP address that shall be used. The entry for the IP address is only necessary if you want to explicitly use a certain TCP device for example if you have several Ethernet or Wireless LAN interfaces installed in your system. If the field is left empty MemBrain will use the system's default LAN interface.

If you uncheck the checkbox above the edit fields then MemBrain will not listen for connection requests at all. Note that in this case other devices will not be able to connect to this MemBrain instance in order to access public neurons. Nevertheless it is still possible for this MemBrain instance to request access to public neurons of other devices.

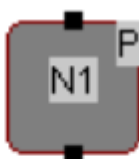
The edit fields of the dialog may be disabled in the following cases:

- The checkbox "Allow operation as server" is not checked
- The WebLink is currently [active](#) (you cannot change the server settings while the WebLink is active, you have to [deactivate](#) it first).

Make Neurons Public

In order to make one or more neurons public just [select](#) them and then choose the menu item <WebLink><Make Neurons Public>.

The neuron(s) will change to look as following



Note that the letter 'P' appeared in the upper right corner of the neuron to indicate that it is now public.

Invoke Extern Neurons

Most operations related especially to extern neurons are performed over the Extern Neurons Manager which can be opened via the menu option <WebLink><Extern Neurons Manager...>.

The following dialog appears.

External Neurons Manager

Name of Host Connection: LOCALHOST Save Connection

IP Address: 127.0.0.1 Port: 2048 Delete Connection

Request Neurons

Connected Device

Device Information String:

View Capabilities

| # | Name | Type |
|---|------|------|
|---|------|------|

Progress:

Reassign To Net Invoke Close

This dialog enables MemBrain to request information about public neurons of other devices such as other MemBrain instances for example. You'll have to specify to which IP address and port MemBrain shall issue the request. Also you can optionally assign a name to the connection and save it to simplify access to it in the future (button <Save Connection>). All saved connections are accessible via the pull down list box by their names.

Saved connections can be deleted with the button named <Delete Connection>. The only connection that cannot be deleted or saved with different settings is the default connection LOCALHOST that points to the local machine (127.0.0.1) and the default port 2048.

After the settings are adjusted you can request the external device for available public neurons by clicking on <Request Neurons>.

If the WebLink is currently not active you will be prompted if you want to activate it now. You will have to click on <Yes> in order to successfully perform a request.

If MemBrain is able to connect to the requested device the dialog will show a progress bar during the request action and then modify to reflect the requested data e.g. as in the following screen shot.

External Neurons Manager

Name of Host Connection:

IP Address: Port:

Connected Device

Device Information String:

| # | Name | Type |
|----|------|--------|
| 1 | B3 | INPUT |
| 2 | B2 | INPUT |
| 3 | B1 | INPUT |
| 4 | B0 | INPUT |
| 5 | 0 | OUTPUT |
| 6 | 1 | OUTPUT |
| 7 | 2 | OUTPUT |
| 8 | 3 | OUTPUT |
| 9 | 4 | OUTPUT |
| 10 | 5 | OUTPUT |

Progress:

You will see a "Device Information String" that is some short identification a MemBrain device can be asked for. In case of MemBrain this is the word "MemBrain" plus the corresponding software version.

You can now easily invoke one or more of the listed neurons as extern neurons into your net by selecting them in the list and clicking on the button <Invoke>. One or more message boxes will appear after the completion of the operation providing information about the result of the action.

Note that it is only possible to invoke a neuron into a net once. If the neuron is already present in the net the invoke operation will fail and you will be informed about that.

Note that the invoked extern neurons will be arranged in your net in the same way as they are arranged in the original net that contains the corresponding public neurons. Nevertheless you can freely move the neurons to where you want after they have been invoked.

There is another button named <Reassign To Net>. This is an option that provides you with the possibility to redefine already invoked neurons in your net with just requested neurons of another device. To do this [select](#) all external neurons that shall be reassigned in your net. You do not have to close the above dialog for that

action as it is non modal. Also select the requested neurons in the list to which the neurons in the net shall be reassigned. Then press the named button. You will be requested to confirm the action and after the operation is complete a message box will inform you about the result.

The criterion by which neurons are identified during reassign operation is the name of the public neuron requested from the external device. This is also the name that had been originally assigned to the corresponding extern neuron when it was invoked the first time. Even if the extern neuron had been renamed after being invoked it still has stored this original name by which it can be identified during the reassign operation. By this means MemBrain can identify which external neuron has to be reassigned to which requested public neuron.

In most cases the reassign functionality will be used in case either the IP address or the port of an external device changes and the net shall be reassigned to be able to communicate with this device again. Or neurons of an already existing net are manually made extern using the command <WebLink><Make Neurons Extern...> and assigned to corresponding public neurons later on using the above described dialog.

If during requesting public neurons the corresponding information could be requested properly then also the button <View Capabilities> has become active. Clicking that button will bring up a window that shows which of the optional MemBrain protocol related features are supported by the connected device:



In case the communication partner is another MemBrain instance certainly all of the available features are supported.

For a description on the meaning of these items please refer to the [WebLink protocol section](#) and also the [teacher editor](#).

Make Neurons Extern

Although extern neurons are normally invoked into a net by use of the [External Neurons Manager](#) dialog it is also possible to manually convert private neurons to extern neurons.

This can be useful if neurons in an already existing and possibly already trained net shall be replaced by extern neurons. To do this choose <WebLink><Make neurons Extern...> and confirm the following request.

Note that you will later on have to reassign these manually created externneurons to really existing public neurons of other nets using the [External Neurons Manager](#). The names that will then be used to reassign the neurons to public neurons of other nets are the names the neurons had at the time they were made extern. This is the case even if the names of the neurons are changed later on. If you want MemBrain to use other names for the reassign operation then [make the neurons private](#), rename them and make them extern manually again.

Extern Neuron Connection State

If an extern neuron is not connected to its public counterpart then it will show the abbreviation "NC" (for "Not Connected") in its lower right corner indicating this:



One possible reason for this is that the Weblink is currently inactive in general. [Activate the Weblink](#) if you want to connect to public neurons. After the Weblink has been activated MemBrain automatically tries to connect all extern neurons to their public counterparts. If this is possible the "NC" icon of the corresponding extern neurons will disappear.

Another possible reason for an NC symbol on an extern neuron is that the corresponding MemBrain service cannot be contacted or does not reply.

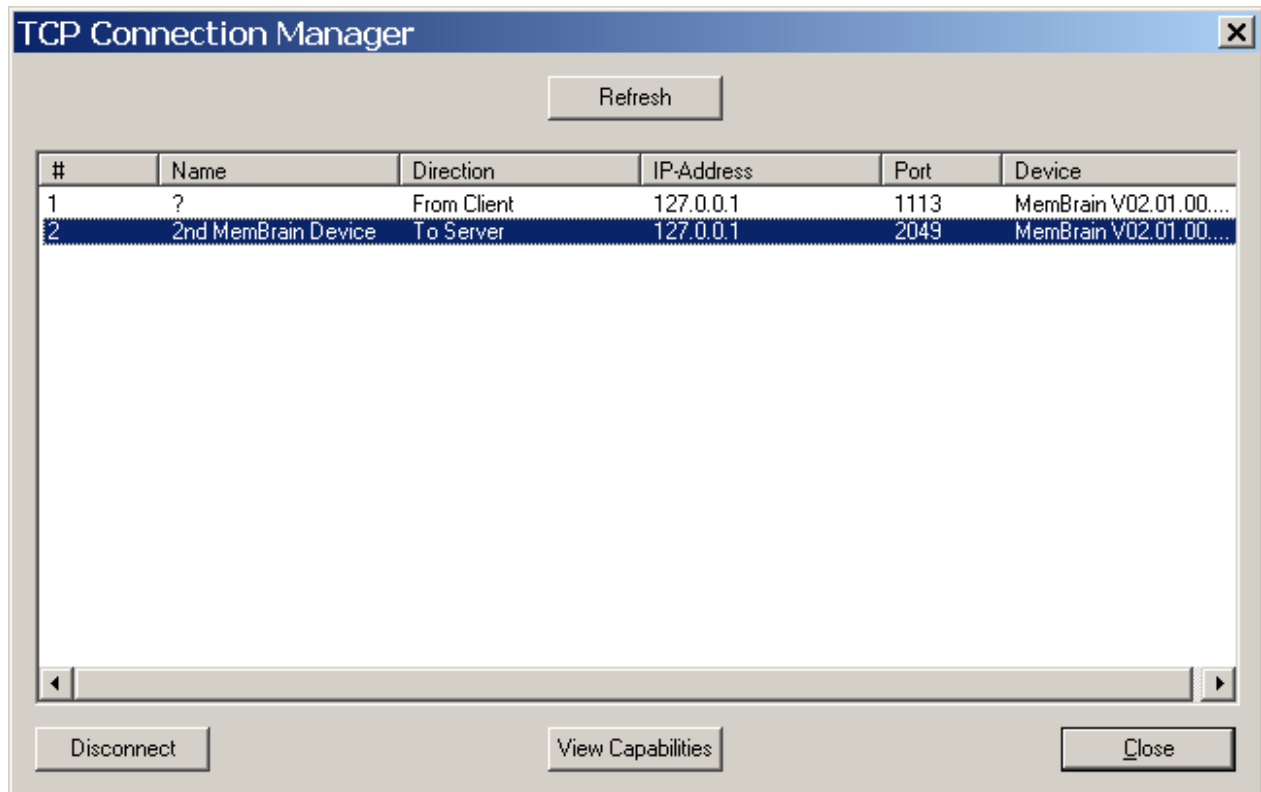
If the Weblink is already activated you can also try to manually initiate another attempt to connect to all public neurons by selecting <WebLink><Reconnect Extern Neurons> from MemBrain's menu.

Make Neurons Private

Either public or extern neurons can be made private, standard neurons by selecting them and choosing <WebLink><Make Neurons Private>.

Manage Active Connections

The menu item <WebLink><Active Connections...> brings up a dialog like this:



It shows a list of all currently active TCP connections of this MemBrain instance together with the main connection details:

List column <Direction>:

Is either of the following.

- To Server
This connection has been requested by this MemBrain instance to the target given by the fields <IP-Address> and <Port>. Through this connection the MemBrain instance accesses public neurons provided by the other device.
- From Client
This connection has been requested by the device given by the fields <IP-Address> and <Port>. Through this connection the other device accesses public neurons provided by this MemBrain instance.

List column <Name>

The name of the connection if it is known. This is the name that has been entered in the Extern Neuron Manager when extern neurons have been invoked for example. Generally a name for a connection only exists if the connection has been requested by the MemBrain instance itself and thus will always be "?" for connections with direction "From Client"

List column <Device>:

The device information string delivered by the connected device

Button <Refresh>:

A click on this button refreshes the list of active connections..

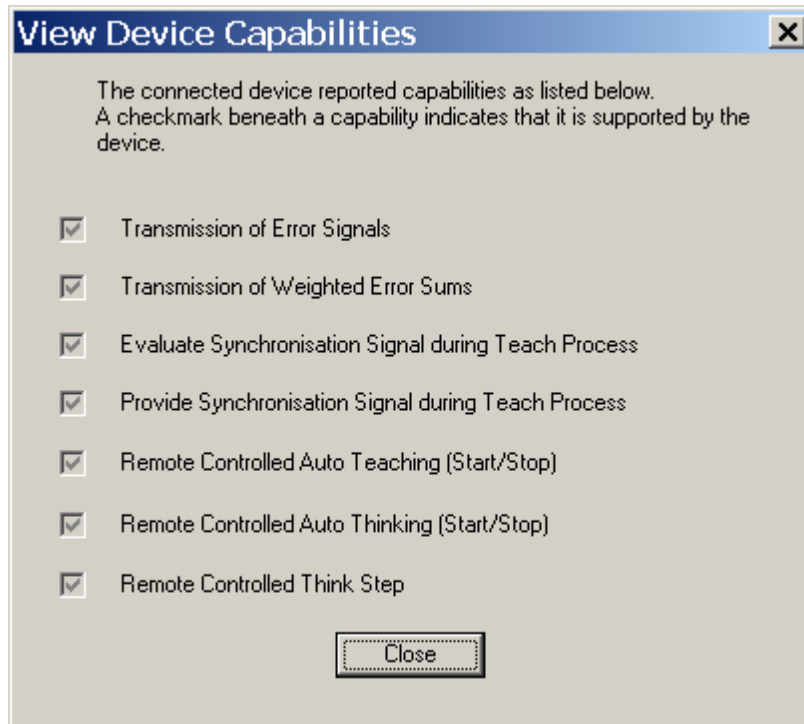
Button <Disconnect>:

You can force a disconnect of all selected list entries by clicking that button. The corresponding

connection(s) will be interrupted immediately without further request.

Button <View Capabilities>:

Clicking that button will bring up a window that shows which of the optional MemBrain protocol related features are supported by the connected device:



Note: This window is also displayed when double clicking on a list entry.

In case the communication partner is another MemBrain instance certainly all of the available features are supported.

For a description on the meaning of these items please refer to the [WebLink protocol section](#) and also the [teacher editor](#).

Remote Auto Think

MemBrain's [auto think](#) process can be remotely started and stopped over the [Weblink](#).

If you want to allow for this you have to activate the option by choosing **<WebLink><Enable Remote Think>**. A check mark has is displayed beneath the menu item if the feature is activated.

If you want the auto think procedure to be performed in synchronization with other MemBrain compatible devices select **<WebLink><Use Weblink Sync during Think Process>**. In this case MemBrain waits for all other connected devices to complete their current think step before it performs another think step of its own.

The Nagle Algorithm

The so called "Nagle Algorithm" is a special algorithm used in TCP communications that shall optimize communication speed by bundling several small message frames together to larger frames as long as

previous smaller frames have not been acknowledged by the recipient on application layer.

This method of transmission performance optimization is mainly useful for slow transmission lines but will sometimes cause a delay on the transmission of small TCP packages.

In case of MemBrain the delay less transmission of small packages is sometimes very important especially during teaching a net. MemBrain does its own package bundling that is optimized to its needs so the Nagle Algorithm is disabled by default as it actually slows down MemBrain's communication performance.

Nevertheless there may be communication scenarios that run better with the algorithm enabled . For this reason MemBrain allows you to enable the Nagle Algorithm by the menu command

<WebLink><Enable Nagle Algorithm>.

Do not use this option unless you are sure of what its effects are. It may slow down your system when working with MemBrain!

Protocol

This chapter provides all details on the protocol used by MemBrain to communicate over it's Weblink. You will need to make your way through this documentation if you want write your own interface software ("driver") to link MemBrain together with proprietary hardware or software.

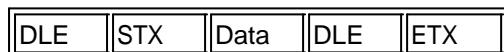
Link Layer

Certainly MemBrain's actual link layer is defined by standard TCP socket communication. Still there is a need to distinguish between logical data packages that form single information frames with dedicated meanings.

This is meant when talking about MemBrain's Link Layer.

MemBrain's Link Layer for Weblink communication is very simple and its only purpose is to enable MemBrain and connected devices to securely distinguish between single information packages that are received via the TCP sockets.

The following framing is used on the link layer:



Start Sequence:

Every data frame starts with the two bytes DLE (= 0x10) and STX (=0x02). This sequence is unique and always indicates a new frame no matter where it occurs within a data stream.

If this sequence is detected while already collecting data of an other frame then the current frame is aborted and all its data is discarded as a result of a transmission error.

Data Stream:

Directly after the Start Sequence the payload portion of the frame follows. Within this stream **every occurrence of the value 0x10 (= DLE) is sent twice** in order to to avoid the data being misinterpreted as a Start or Stop Sequence in case the next data byte is an STX or an ETX. The net data portion is defined to be 100 bytes maximum. Thus in the border line case of all data being 0x10 the resulting data stream could get 200 bytes long.

Stop Sequence:

As a unique Stop Sequence the bytes DLE and ETX (=0x03) are used. If this byte sequence is detected in the data stream then the frame is complete and can be processed by the [Application Layer](#).

As a result of the above said the overall size of a Link Layer frame is always between 5 and 204 bytes. (2

Bytes Start Sequence, 2 Bytes Stop Sequence, between 1 and 200 bytes of raw data resulting in 1 to 100 bytes payload data for the frame). Nevertheless all currently used commands are smaller in size (see [here](#) for a list of commands). The overhead is reserved for future use.

TCP configuration settings:

When writing your own TCP "driver" for some MemBrain compatible device you should disable the [Nagle Algorithm](#) of your TCP stack and use optimized message packaging instead as described [here](#).

Programmers Hint:

The Nagle Algorithm is disabled by using the socket option **TCP_NODELAY**.

Application Layer

After a data frame has been extracted by the [Link Layer](#) MemBrain's application layer interprets and processes the received data. Every received data package from the Link Layer corresponds to one dedicated command of the application layer.

Framing

MemBrain supports two types of application layer frames, one that uses additional address information to access dedicated target neurons and one to exchange information that is independent from neuron addresses:

Addressed Frame:

| Data Type | Neuron Address | Neuron Address | Neuron Address | Neuron Address | Command | Data (optional) |
|-----------|----------------|----------------|----------------|----------------|---------|-----------------|
| | Low LSB | Low MSB | High LSB | High MSB | | |

The Neuron Address is an unsigned 32 bit value uniquely identifying the *public* neuron within its net. This value can be retrieved for every public neuron using [common service](#) commands and will not change during the lifetime of a neural net. Thus it has to be requested only once when an external neuron is [invoked](#) into a net for the first time.

Non-Addressed Frame:

| Data Type | Command | Data (optional) |
|-----------|---------|-----------------|
| | | |

The first byte, the "Data Type" decides which of the frames is used (see below).

Data Type:

The Data Type is always one byte and can be one of the following.

- 0 = [Common Services](#). Uses non-addressed frames only

- 1 = [Command from a public to an extern neuron](#). Uses addressed frames only.
- 2 = [Command from an extern to a public neuron](#). Uses addressed frames only.

Command:

The command itself is always one byte. This command also determines if additional data is used together with the command or not. The complete set of commands is described in the following chapters.

Data:

A stream of additional data bytes can follow the command. The formatting of that data is dependent on the command byte and described together with every command.

Data Encoding

This section gives some general information on how data is encoded/decoded when being transmitted on application layer level.

Byte Order:

As a general rule, data representing values larger than 8 bit is transmitted **LSB first**.

Floating Point Numbers:

Floating Point numbers generally are converted to Fixed Point numbers before being transmitted over the Weblink. This implies some loss of precision but keeps the protocol simple so that also very resource limited systems can be connected to MemBrain without the need of implementing floating point arithmetic. Especially for micro controller based interface hardware floating point arithmetic is a very resource consuming issue with respect to memory usage and performance.

There are two conversions used by MemBrain:

FIXED

This conversion is used when the data range to be transmitted exceeds -1 respectively 1. For conversion the floating point number is multiplied by 10000 and transmitted as an INT32 value (4 bytes long). Thus the range for the values can be -214747.0000 to +214748.0000 or in integer notation -2147470000 to +2147480000

Consequently, every value that is received in the FIXED format has to be divided by 10000 if it shall be converted back to a floating point number again.

FIXED_M1_P1

This fixed point conversion is used for all values that cannot exceed the range of -1 to 1.

In this case a smaller representation with 16 bits can be chosen for the conversion: The corresponding floating point number is multiplied by 10000 and then transmitted as an INT16 value ranging from -10000 to +10000 or interpreted as floating point number from -1.0000 to + 1.0000.

Consequently, every value that is received in the FIXED_M1_P1 format has to be divided by 10000 if it shall be converted back to a floating point number again.

Timeouts

During communication between MemBrain devices a general timeout of 30 seconds is used. After 30 s it is assumed that there will be no reply to a command anymore. Nevertheless all devices must be able to deal with a reply that arrives after more than 30 s and if it is just to discard it.

Command Reference

This section lists and describes all of the commands used by MemBrain's TCP based protocol.

Standard Commands

This section and its sub sections list all commands that are correlated to the standard MemBrain functionality i.e. exchange of information about published neurons, requests of input signals, activation and output signals etc.

Common Services

This section lists and describes all standard commands related to common services i.e. all standard commands that are not addressed to a specific neuron.

Note that all these commands use [non-addressed frames](#)

SVR_REQ_DEVICE

Purpose: Request Device Information String

Command Value: 64

Parameters: None

Description: Request a string from the connected device that gives information about the device. If this command is sent to MemBrain the replied string is "MemBrain Vxx.xx.xx.xx" where xx.xx.xx.xx is the software version number.

Compatibility: Mandatory for every MemBrain compatible device

Usage: MemBrain sends this command every time a new TCP connection to a device has been established or when information about public neurons from a connected device is requested.

SVR_DEVICE

Purpose: Transmit Device Information String

Command Value: 1

Parameters: Device information string, NULL terminated, 30 chars max.

Description: Reply to SVR_REQ_DEVICE
MemBrain sends "MemBrain Vxx.xx.xx.xx" as string where xx.xx.xx.xx is the software version number.

Compatibility: Mandatory for every MemBrain compatible device

SVR_REQ_DEVICE_CAPS

Purpose: Request Device Capabilities

Command Value: 65

Parameters: None

Description: Request information about the features supported by the connected device.

Compatibility: Mandatory for every MemBrain compatible device

Usage: MemBrain sends this command every time a new TCP connection to a device has been established or when

information about public neurons from a connected device is requested.

SVR_DEVICE_CAPS

Purpose: Report Device Capabilities

Command Value: 2

Parameters: Device capabilities, 4Bytes interpreted as one unsigned 32bit value.
Every possible capability is represented by one bit within this value.

The following capabilities are defined.

- DEV_CAPS_PROVIDE_TEACH_WEBLINK_SYNC = 0x00000001
- DEV_CAPS_EVALUATE_TEACH_WEBLINK_SYNC = 0x00000002
- DEV_CAPS_SUPP_ERR_SIG = 0x00000004
- DEV_CAPS_SUPP_WGHT_ERR_SUM = 0x00000008
- DEV_CAPS_SUPP_WEBLINK_AUTO_TEACH_REMOTE = 0x00000010
- DEV_CAPS_SUPP_WEBLINK_AUTO_THINK_REMOTE = 0x00000020
- DEV_CAPS_SUPP_WEBLINK_THINK_STEP_REMOTE = 0x00000040
- DEV_CAPS_PROVIDE_THINK_WEBLINK_SYNC = 0x00000080
- DEV_CAPS_EVALUATE_THINK_WEBLINK_SYNC = 0x00000100

All remaining bits are reserved for future use and should be set to 0 when device caps are reported.

For a description on the meaning of the different device capabilities, see [here](#).

Description: Reply to SVR_REQ_DEVICE_CAPS
MemBrain sends the ORED representation of all possible capabilities, i.e. the value 0x0000007F.

Compatibility: Mandatory for every MemBrain compatible device

SVR_REQ_PN_COUNT

Purpose: Request number of published neurons

Command Value: 66

Parameters: None

Description: Requests the number of available public neurons from a connected device

Compatibility: Mandatory for every MemBrain compatible device

Usage:

MemBrain sends this command every time when information about public neurons from a connected device is requested in order to [invoke](#) them as external neurons into the net or to [reassign](#) them to existing external neurons.

SVR_PN_COUNT

NOTE: The parameters for this command have changed since MemBrain version

| | |
|----------------|---|
| 02.01.01.03: | The number of neurons that can be published has been increased from unsigned 16bit to (signed) 32 bit. |
| Purpose: | Report number of published neurons |
| Command Value: | 3 |
| Parameters: | 4 bytes, interpreted as signed 32bit value (only positive values are allowed). Thus there is a theoretical maximum of 2147483648 neurons a MemBrain device can publish. |
| Description: | Reply to SVR_REQ_PN_COUNT |
| Compatibility: | Mandatory for every MemBrain compatible device |

SVR_REQ_PN

| | |
|----------------|--|
| NOTE: | This command isn't supported anymore since MemBrain version 02.01.01.03 It has been replaced by SVR_REQ_PN_ALL |
| Purpose: | Request details about one public neuron |
| Command Value: | 67 |
| Parameters: | 2 bytes, interpreted as unsigned 16bit value. This is the index of the neuron the details are requested for and must be a value between 0 and the number returned by SVR_PN_COUNT (see above). |
| Description: | Requests details about one public neuron from a connected device |
| Compatibility: | Mandatory for every MemBrain compatible device |
| Usage: | MemBrain sends this command for every public neuron every time when information about public neurons from a connected device is requested in order to <u>invoke</u> them as external neurons into the net or to <u>reassign</u> them to existing extern neurons. |

SVR_REQ_PN_ALL

| | |
|----------------|---|
| NOTE: | This command replaces the command SVR_REQ_PN since MemBrain version 02.01.01.03 |
| Purpose: | Request details about all public neurons |
| Command Value: | 68 |
| Parameters: | none |
| Description: | Requests details about all public neurons from a connected device |
| Compatibility: | Mandatory for every MemBrain compatible device |
| Usage: | MemBrain sends this command only once for all every public neurons every time when information about public neurons from a connected device is requested in order to <u>invoke</u> them as external neurons into the net or to <u>reassign</u> them to existing extern neurons. |

SVR_PN

NOTE: This command parameters have changed since MemBrain version 02.01.01.03:
The neuron index is four byte now instead of 2 in former versions

Purpose: Report details about one public neuron

Command Value: 4

Parameters: See table below

Description: Reply to SVR_REQ_PN

Compatibility: Mandatory for every MemBrain compatible device

| Parameter | Size | Description |
|---------------------------|---------|---|
| Neuron Index | 4 bytes | INT32 value indicating the index of the neuron. Always between 0 and the number returned by SVR_PN_COUNT |
| Unique Neuron ID in net | 4 bytes | UINT32 value to be used as address with all commands dedicated to this public neuron |
| Neuron Type | 1 byte | 0 = INPUT, 1 = HIDDEN, 2 = OUTPUT |
| Activation Function | 1 byte | 0 = LOGISTIC 1 = IDENTICAL 2 = IDENTICAL_0_T O_1 3 = TAN_HYP 4 = BINARY 5 = MIN_EUCLID_DIST |
| Activation Threshold | 4 byte | INT32 value in FIXED representation |
| Activation Sustain Factor | 2 byte | INT16 value in FIXED_M1_P1 representation |
| Output Fire Level | 1 byte | 0 = "1" 1 = |

| | | |
|----------------------|--------------|---|
| | | "ACTIVATION" |
| Output Recovery Time | 4 bytes | UINT32 value, interpreted as number of think steps |
| Lower Fire Threshold | 4 byte | INT32 value in FIXED representation |
| Upper Fire Threshold | 4 byte | INT32 value in FIXED representation |
| Screen Position X | 2 byte | X-Coordinate of the neuron's center on the screen. UINT16 value counted from left to right |
| Screen Position Y | 2 byte | Y-Coordinate of the neuron's center on the screen. UINT16 value counted from top to bottom |
| Neuron Name | 31 bytes max | NULL terminated string representing the neuron's name. 30 chars max |

SVR_AUTO_THINK_STARTED

Purpose: Report that the auto think process has been started

Command Value: 9

Parameters: None

Description: Used for remotely starting other MemBrain devices during auto think process.

Compatibility: Optional. Recommended when the device's auto think process can be started/stopped.
See also [Device Capabilities](#) for more information.

SVR_AUTO_THINK_STOPPED

Purpose: Report that the auto think process has been stopped

Command Value: 10

| | |
|----------------|--|
| Parameters: | None |
| Description: | Used for remotely stopping other MemBrain devices during auto think process. |
| Compatibility: | Optional. Recommended when the device's auto think process can be started/stopped. See also Device Capabilities for more information. |

SVR_THINK_STEP

| | |
|----------------|--|
| Purpose: | Report that a manually triggered think step is performed. |
| Command Value: | 11 |
| Parameters: | None |
| Compatibility: | Optional. Recommended when the device supports single think steps. See also Device Capabilities for more information. |

SVR_THINK_STEP_COMPLETE

| | |
|----------------|--|
| Purpose: | Report that a think step has been completed |
| Command Value: | 12 |
| Parameters: | None |
| Description: | Used for synchronisation between MemBrain devices when thinking. |
| Compatibility: | Optional. Recommended when the device implements its own thinking process. See also Device Capabilities for more information. |

SVR_NOT_THINKING

| | |
|----------------|--|
| Purpose: | Response to SVR_THINK_STEP_COMPLETE if not currently thinking |
| Command Value: | 13 |
| Parameters: | None |
| Description: | Used for synchronisation between MemBrain devices when thinking but the corresponding MemBrain device is not currently thinking. Prevents MemBrain from waiting on that device. |
| Compatibility: | Optional. Recommended when the device implements its own thinking process. See also Device Capabilities for more information. |

Device Capabilities

This chapter describes the meaning of the device capabilities that MemBrain requests from every connected device.

A list of the values of all device capabilities requested from a connected device can be displayed in MemBrain. For details on how to do this, see [here](#). and [here](#).

DEV_CAPS_PROVIDE_TEACH_WEBLINK_SYNC

During the teach process MemBrain supports synchronisation by means of certain commands (SVR_TEACH_STEP_COMPLETE resp. SVR_NOT_TEACHING, see [here](#) for details).

By the exchange of these commands a device can tell all other connected devices that it has finished the last teach step. If MemBrain's currently active teacher is configured to use this weblink sync mechanism it will wait for all connected devices to finish their teach step before it will start another teach step on its own. The first teach step is always performed without waiting for other devices in order to prevent a deadlock situation.

Thus all devices that support this mechanism will do the same number of teach steps within the same time interval. If a connected device supports that feature it should report this capability. Otherwise MemBrain will never wait for that device finishing its teach step because it by default assumes that the device does not support the feature.

Note: If a device reports DEV_CAPS_PROVIDE_TEACH_WEBLINK_SYNC it has to implement both SVR_TEACH_STEP_COMPLETE and SVR_NOT_TEACHING to be fully compatible with MemBrain.

DEV_CAPS_EVALUATE_TEACH_WEBLINK_SYNC

If a device wants to receive the commands SVR_TEACH_STEP_COMPLETE resp. SVR_NOT_TEACHING from MemBrain it has to report this capability. Otherwise MemBrain will not send these commands to the device although it still evaluates these commands when received from the device (see above).

DEV_CAPS_SUPP_ERR_SIG

Report this capability if your device supports requests of error signals to its public neurons (command EXT_NR_REQ_ERR_SIG, see [here](#)).

If this capability is set then MemBrain may ask the public neurons of your device for error signals during the teach process. Error signals are values propagated backwards through the net from the output layer through the hidden layers towards the input layer when the net is taught by a so-called Backpropagation based teacher which currently is the case for all teachers implemented by MemBrain.

DEV_CAPS_SUPP_WGHT_ERR_SUM

If a device supports the request for weighted error sums to its extern neurons then it should report this capability.

Weighted error sums are requested during the teach process by the public neurons of MemBrain from all extern neurons that connect to the corresponding public neurons. A weighted error sum is the sum of the error signals of all neurons connected to the output of a neuron where each summand is weighted by the corresponding link weight over which the neuron is connected.

The weighted error sum is required during calculation of the error signal of a neuron. Thus to calculate the error signal of a public neuron the weighted error sums of all connected extern neurons have to be requested because the output of the public neuron may be connected to other neurons within other nets through its extern counterparts.

DEV_CAPS_SUPP_WEBLINK_AUTO_TEACH_REMOTE

If a device wants to receive SVR_TEACHER_STARTED resp. SVR_TEACHER_STOPPED commands from MemBrain it has to report this capability.

By the named commands the teachers of all MemBrain compatible devices that are connected over the TCP WebLink can be started/stopped synchronously fromout any device that supports these commands. In order to do this the following two behaviours have to be implemented for each device

- When receiving a SVR_TEACHER_STARTED command from any connected device then start the own teacher, too. Correspondingly when a SVR_TEACHER_STOPPED command is received, then stop the own teacher.
- Send a SVR_TEACHER_STARTED command to all connected other devices every time the teaching process is started, no matter if the teacher has been started manually or by reception of a SVR_TEACHER_STARTED command . Correspondingly send a SVR_TEACHER_STOPPED command when the teacher is stopped.

DEV_CAPS_SUPP_WEBLINK_AUTO_THINK_REMOTE

If a device wants to receive SVR_AUTO_THINK_STARTED resp. SVR_AUTO_THINK_STOPPED commands from MemBrain it has to report this capability.

By the named commands the think process (continuous simulation of the network) of all MemBrain compatible devices that are connected over the TCP WebLink can be started/stopped synchronously fromout any device that supports these commands. In order to do this the following two behaviours have to be implemented for each device

- When receiving a SVR_AUTO_THINK_STARTED command from any connected device then start the own auto think process, too. Correspondingly when a SVR_AUTO_THINK_STOPPED command is received, then stop the think process.
- Send a SVR_AUTO_THINK_STARTED command to all connected other devices every time the auto think process is started, no matter if the process has been started manually or by reception of a SVR_AUTO_THINK_STARTED command . Correspondingly send a SVR_AUTO_THINK_STOPPED command when the auto think process is stopped.

DEV_CAPS_SUPP_WEBLINK_THINK_STEP_REMOTE

If a device wants to receive SVR_THINK_STEP commands from MemBrain it has to report this capability.

By the named command the process of performing one think step (one calculation step over all alyers of the net) of all MemBrain compatible devices that are connected over the TCP WebLink can be performed synchronously fromout any device that supports this command.

In order to do this the following two behaviours have to be implemented for each device

- When receiving a SVR_THINK_STEP command from any connected device then perform a think step on its own.
- When performing a **manually started** think step then send a SVR_THINK_STEP command to all connected devices. Do not send a SVR_THINK_STEP when performing a think step initiated by the reception of a SVR_THINK_STEP command!

DEV_CAPS_PROVIDE_THINK_WEBLINK_SYNC

During the teach process MemBrain supports synchronisation by means of certain commands (SVR_THINK_STEP_COMPLETE resp. SVR_NOT_THINKING, see [here](#) for details).

By the exchange of these commands a device can tell all other connected devices that it has finished the last think step. If MemBrain is currently configured to use this weblink sync mechanism it will wait for all connected devices to finish their think step before it will start another think step on its own. The first think step is always performed without waiting for other devices in order to prevent a deadlock situation. Thus all devices that support this mechanism will do the same number of think steps within the same time interval. If a connected device supports that feature it should report this capability. Otherwise MemBrain will never wait for that device finishing its think step because it by default assumes that the device does not

support the feature.

Note: If a device reports DEV_CAPS_PROVIDE_THINK_WEBLINK_SYNC it has to implement both SVR_THINK_STEP_COMPLETE and SVR_NOT_THINKING to be fully compatible with MemBrain.

DEV_CAPS_EVALUATE_THINK_WEBLINK_SYNC

If a device wants to receive the commands SVR_THINK_STEP_COMPLETE resp. SVR_NOT_THINKING from MemBrain it has to report this capability. Otherwise MemBrain will not send these commands to the device although it still evaluates these commands when received from the device (see above).

Commands From Public Neurons

This section lists and describes all commands from public neurons to their extern counterparts.

Note that all these commands use [addressed frames](#) i.e. the unique ID of every public neuron is always included in the command frame. This ID can be retrieved from every public neuron using the common service command SVR_REQ_PN as described [here](#).

PUB_NR_ACT_OUT

| | |
|----------------|--|
| Purpose: | Report Activation and output signal of the public Neuron |
| Command Value: | 3 |
| Parameters: | 4 bytes representing two FIXED_M1_P1 values. First value is activation, second one is output value. |
| Description: | Reply to EXT_NR_REQ_ACT_OUT or EXT_NR_REQ_CHNG_ACT_OUT. Transmits the current activation and output signal of a public neuron to its extern counterpart. |
| Compatibility: | Mandatory for every MemBrain compatible device |

Usage:

MemBrain uses the commands EXT_NR_REQ_ACT_OUT resp. EXT_NR_REQ_CHNG_ACT_OUT frequently during think and teach process and expects a reply PUB_NR_ACT_OUT to them.

PUB_NR_REQ_INP

| | |
|----------------|---|
| Purpose: | Request Input Signal Sum of the extern counterpart neuron |
| Command Value: | 64 |
| Parameters: | none |
| Description: | To perform its calculation properly every public neuron that is not an input neuron needs to know what signals are currently applied to its input. By use of this command these values are requested from the extern counterparts of the public neuron. |
| Compatibility: | Mandatory for every MemBrain compatible device |

Usage:

Every MemBrain compatible device has to repetitively request the input signals to every one of its public neurons. In order to do so every public neuron has to repetitively ask all its extern counterparts for their current input signal sum.

To keep the data flow acceptably low a combination of two commands is used to realise that:

PUB_NR_REQ_CHNG_INP and PUB_NR_REQ_INP While PUB_NR_REQ_CHNG_INP results in a reply only upon a change of the input signals to the extern neurons, PUB_NR_REQ_INP expects an immediate reply. Every device should use only PUB_NR_REQ_CHNG_INP commands unless one of these commands [times out](#).

Upon time out of the PUB_NR_REQ_CHNG_INP command a PUB_NR_REQ_INP shall be sent to force a request of the input signals.

The reply to PUB_NR_REQ_CHNG_INP or PUB_NR_REQ_INP can be either EXT_NR_INP or EXT_NR_NO_INPUTS.

PUB_NR_REQ_CHNG_INP

Purpose: Request Change of the Input Signal Sum of the extern counterpart neuron

Command Value: 65

Parameters: none

Description: To perform its calculation properly every public neuron needs to know what signals are currently applied to its input. By use of this command a change in these values is requested from the extern counterparts of the public neuron.

The command requests the extern neuron to transmit its input signal sum once it changes with respect to the last transmitted values.

The reply to this command occurs once the input signal sum of the corresponding extern counterpart neuron changes or immediately if the change has already happened.

If there is no change the command will [time out](#).

Compatibility: Mandatory for every MemBrain compatible device

Usage:

Every MemBrain compatible device has to repetitively request the input signals to every one of its public neurons. In order to do so every public neuron has to repetitively ask all its extern counterparts for their current input signal sum.

To keep the data flow acceptably low a combination of two commands is used to realise that:

PUB_NR_REQ_CHNG_INP and PUB_NR_REQ_INP

While PUB_NR_REQ_CHNG_INP results in a reply only upon a change of the input signals with respect to the last transmitted values, PUB_NR_REQ_INP expects an immediate reply. Every device should use only PUB_NR_REQ_CHNG_INP commands unless one of these commands

[times out](#).

Upon time out of the PUB_NR_REQ_CHNG_INP command a PUB_NR_REQ_INP shall be sent to the corresponding extern neuron in order to force a request of the input signals.

The reply to PUB_NR_REQ_CHNG_INP or PUB_NR_REQ_INP can be either EXT_NR_INP or EXT_NR_NO_INPUTS.

PUB_NR_PING

Purpose: Dummy command to check if an extern neuron is present over the Weblink

Command Value: 254

Parameters: none

Description: If it is necessary to determine if a public neuron is able to access one of its extern counterparts this command can be sent.

If the extern neuron is present it will answer with EXT_NR_PONG

Compatibility: MemBrain does not use this command as there is no indicator in MemBrain that shows the user the connection state to extern neurons. Nevertheless if a device invokes extern neurons it should implement the command to be compatible with future versions that may use the command.

Usage:

Not used by MemBrain but MemBrain replies correctly with EXT_NR_PONG

PUB_NR_PONG

Purpose: Reply to EXT_NR_PING

Command Value: 255

Parameters: none

Description: An extern neuron can check if its public counterpart is accessible over the Weblink. If this is the case the public neuron shall reply with PUB_NR_PONG.

Compatibility: Mandatory for every MemBrain compatible device that publishes neurons.

Usage:

MemBrain uses EXT_NR_PING commands to check the accessibility of public neurons and in order to register the extern neurons at their public counterparts.

Commands From Extern Neurons

This section lists and describes all commands from extern neurons to their public counterparts.

Note that all these commands use [addressed frames](#) i.e. the unique ID of every public neuron is always included in the command frame. This ID can be retrieved from every public neuron using the common service command SVR_REQ_PN as described [here](#).

EXT_NR_INP

| | |
|----------------|--|
| Purpose: | Report the sum of the input signals of the extern neuron |
| Command Value: | 1 |
| Parameters: | 4 bytes representing one FIXED value |
| Description: | Reply to PUB_NR_REQ_INP or PUB_NR_REQ_CHNG_INP. Transmits the current input signal sum of an extern neuron to its public counterpart. |
| Compatibility: | Mandatory for every MemBrain compatible device. |

Usage:

MemBrain uses this command frequently when calculating the net output: Every public neuron may have additional inputs which are connected to its extern counterpart(s). Thus every public neuron must ask all its extern counterparts for input signals when its activation and output signal shall be calculated.

EXT_NR_NO_INPUTS

| | |
|----------------|--|
| Purpose: | Report that an extern neurons has no inputs connected to it. |
| Command Value: | 2 |
| Parameters: | none |
| Description: | Reply to PUB_NR_REQ_INP or PUB_NR_REQ_CHNG_INP. Informs a public neuron that the corresponding extern counterpart has no input signals connected to it. |
| Compatibility: | Mandatory for every MemBrain compatible device. |

Usage:

If the input connector of an extern neuron is open, i.e. there are no input links connected to it then it has to respond with EXT_NR_NO_INPUTS upon requests PUB_NR_REQ_INP or PUB_NR_REQ_CHNG_INP from its public counterpart. Note that this is not the same as responding EXT_NR_INP with an input sum of 0! A neuron with open inputs calculates its activation different from a neuron with an input signal.

EXT_NR_REQ_ACT_OUT

| | |
|----------------|---|
| Purpose: | Request the Activation and the Output Signal of the public Neuron |
| Command Value: | 68 |
| Parameters: | none |

Description: Requests the current activation and output signal of a public neuron

Compatibility: Mandatory for every MemBrain compatible device

Usage:

MemBrain uses the commands EXT_NR_REQ_ACT_OUT resp. EXT_NR_REQ_CHNG_ACT_OUT frequently during think and teach process and expects a reply PUB_NR_ACT_OUT to them.

EXT_NR_REQ_CHNG_ACT_OUT

Purpose: Request the next change in Activation or Output Signal of the public Neuron

Command Value: 69

Parameters: none

Description: Requests the public neuron to transmit its activation and output signal once a change of one of the signals occurs with respect to the last transmitted values.

If there is no change the command will [time out](#).

Compatibility: Mandatory for every MemBrain compatible device

Usage:

MemBrain uses the commands EXT_NR_REQ_ACT_OUT resp. EXT_NR_REQ_CHNG_ACT_OUT frequently during think and teach process and expects a reply PUB_NR_ACT_OUT to them.

.

EXT_NR_PING

Purpose: Dummy command to check if a public neuron is present over the Weblink

Command Value: 254

Parameters: none

Description: If it is necessary to determine if an extern neuron is able to access its public counterpart this command can be sent.
If the public neuron is present it will answer with PUB_NR_PONG

Compatibility: Mandatory for every MemBrain compatible device that publishes neurons.

Usage:

MemBrain uses EXT_NR_PING commands to check the accessibility of public neurons and in order to register the extern neurons at their public counterparts.

EXT_NR_PONG

Purpose: Reply to PUB_NR_PING

Command Value: 255

Parameters: none

| | |
|----------------|---|
| Description: | A public neuron can check if one of its public counterparts is accessible over the Weblink using the command PUB_NR_PING. If this is the case the extern neuron shall reply with EXT_NR_PONG. |
| Compatibility: | MemBrain does not use PUB_NR_PING as there is no indicator in MemBrain that shows the user the connection state to extern neurons. Nevertheless if a device invokes extern neurons it should implement the command to be compatible with future versions that may use the command. |
| Usage: | MemBrain replies EXT_NR_PONG upon reception of a PUB_NR_PING |

EXT_NR_ACT

| | |
|----------------|--|
| Purpose: | Report the activation of the the extern neuron. |
| Command Value: | 4 |
| Parameters: | 2 bytes representing one FIXED_M1_P1 value. |
| Description: | This command is used to set the activation of a public neuron by use of one of its extern counterparts. It is needed if input data shall be applied to the public neurons of a net by remote control. Thus it is not a reply to a request but send on initiative of the external neuron. |
| Compatibility: | Optional. Can be used by software in order to remote control MemBrain. |

Usage:
MemBrain uses this command when the activation of an external neuron is set manually (e.g. by selecting the neuron and typing "1" on the keypad. The public counterpart will overtake this activation when it receives the command. Can be used to remote controlled apply input patterns to neuron nets in MemBrain.

Teacher Commands

This section and its sub sections list all commands that are correlated to MemBrain's teaching functionality i.e. exchange of error signals etc.

If a driver does not implement an own teaching process then the commands of these chapters are not relevant for the driver.

Common Services

This section lists and describes all teacher related common service commands i.e. all teacher related commands that are not addressed to a specific neuron.

Note that all these commands use [non-addressed frames](#)

SVR_TEACH_STEP_COMPLETE

| | |
|----------------|--|
| Purpose: | Report that a teacher step has been completed |
| Command Value: | 5 |
| Parameters: | None |
| Description: | Used for synchronisation between MemBrain devices when teaching. |
| Compatibility: | Optional. Recommended when the device implements its own teaching process. See also Device Capabilities for more information. |

SVR_NOT_TEACHING

| | |
|----------------|--|
| Purpose: | Response to SVR_TEACH_STEP_COMPLETE if not currently teaching |
| Command Value: | 6 |
| Parameters: | None |
| Description: | Used for synchronisation between MemBrain devices when teaching but the corresponding MemBrain device is not currently teaching. Prevents MemBrain from waiting on that device. |
| Compatibility: | Optional. Recommended when the device implements its own teaching process. See also Device Capabilities for more information. |

SVR_TEACHER_STARTED

| | |
|----------------|--|
| Purpose: | Report that the teacher has been started (in auto teach mode) |
| Command Value: | 7 |
| Parameters: | None |
| Description: | Used for remotely starting other MemBrain devices during teaching |
| Compatibility: | Optional. Recommended when the device implements its own teaching process. See also Device Capabilities for more information. |

SVR_TEACHER_STOPPED

| | |
|----------------|--|
| Purpose: | Report that the teacher has been stopped (in auto teach mode) |
| Command Value: | 8 |
| Parameters: | None |
| Description: | Used for remotely stopping other MemBrain devices during teaching |
| Compatibility: | Optional. Recommended when the device implements its own teaching process. See also Device Capabilities for more information. |

Commands From Public Neurons

This section lists and describes all teacher related commands from public neurons to their extern counterparts.

Note that all these commands use [addressed frames](#) i.e. the unique ID of every public neuron is always included in the command frame.

This ID can be retrieved from every public neuron using the common service command SVR_REQ_PN as described [here](#).

PUB_NR_ERR_SIG

| | |
|----------------|--|
| Purpose: | Report Error Signal of the public neuron |
| Command Value: | 4 |
| Parameters: | 4 bytes representing one FIXED value for the Error Signal. |
| Description: | Reply to EXT_NR_REQ_ERR_SIG. Transmits the current error signal of a public neuron to its extern counterpart. |
| Compatibility: | Optional. Recommended when the device implements its own teaching process. See also Device Capabilities for more information. |

Usage:

If the connected device supports the feature (see [Device Capabilities](#) for more information) MemBrain uses the command EXT_NR_REQ_ERR_SIG frequently during teach process and expects a reply PUB_NR_ERR_SIG to it.

PUB_NR_REQ_ERR_SUM

| | |
|----------------|--|
| Purpose: | Request weighted output error sum of the extern counterpart. |
| Command Value: | 66 |
| Parameters: | none |
| Description: | During teaching the Error Signal of every neuron has to be determined. Thus every public neuron has to request the weighted error sum of every one of its extern counterparts. |
| Compatibility: | Optional. Recommended when the device implements its own backpropagation based teaching process. See also Device Capabilities for more information. |

Usage:

Commands From Extern Neurons

This section lists and describes all teacher related commands from extern neurons to their public counterparts.

Note that all these commands use [addressed frames](#) i.e. the unique ID of every public neuron is always included in the command frame.

This ID can be retrieved from every public neuron using the common service command SVR_REQ_PN as described [here](#).

EXT_NR_WGHT_ERR_SUM

| | |
|----------------|--|
| Purpose: | Report the weighted error sum of the extern neuron |
| Command Value: | 3 |
| Parameters: | 4 bytes representing one FIXED value |
| Description: | Reply to PUB_NR_REQ_ERR_SUM. Transmits the current weighted error sum of the extern neuron to its public counterpart. |
| Compatibility: | Optional. Recommended when the device implements its own teaching process. See also Device Capabilities for more information. |

Usage:

MemBrain uses this command frequently during teaching to request the weighted error sum of all extern neurons linked to a public neuron.

The weighted error sum is the sum over the error signals of all neurons connected to the output of a neuron where every summand is multiplied with the weight of the link that connects to the corresponding neuron.

EXT_NR_REQ_ERR_SIG

| | |
|----------------|--|
| Purpose: | Request the Error Signal of a public neuron |
| Command Value: | 70 |
| Parameters: | none |
| Description: | Requests the current error signal of a public neuron |
| Compatibility: | Optional. Recommended when the device implements its own teaching process. See also Device Capabilities for more information. |

Usage:

MemBrain uses the commands EXT_NR_REQ_ACT_OUT resp. EXT_NR_REQ_CHNG_ACT_OUT frequently during think and teach process and expects a reply PUB_NR_ACT_OUT to them.

Communication Strategies

This chapter summarizes the communication flow between MemBrain and another instance of MemBrain or some MemBrain compatible device. It is provided to simplify the process of writing a driver that interfaces between MemBrain and other software or hardware.

All communication listed here covers the commands that are mandatory for every MemBrain compatible device. There are [additional commands](#) used during teaching process if the connected devices report that they support the corresponding [capabilities](#). This communication is not explained here as it is only necessary if the device implements its own teaching algorithms. This probably does not apply for most of the applications.

The communication partners are named M1 resp. M2 in the following listings

Command flow when M1 requests public neurons from M2:

```

M1:  SVR_REQ_DEVICE      (Request Device Information String)
M2:  SVR_DEVICE          (Transmit Device Information String)

M1:  SVR_REQ_DEVICE_CAPS (Request Device Capabilities)
M2:  SVR_DEVICE_CAPS     (Transmit Device Capabilities)

M1:  SVR_REQ_PN_COUNT    (Request number of published neurons)
M2:  SVR_PN_COUNT        (Transmit number of published neurons <count>)

M1:  SVR_REQ_PN_ALL      (Request details about all the public neurons)

for (int i = 0; i < count; i++)
{
    M2:  SVR_PN           (Transmit details about the public neuron i)
}

```

Command flow when requested neurons are invoked by M1 as extern neurons:

```

for every invoked neuron
{
    M1:  EXT_NR_PING      (Dummy command to test is public neuron can be accessed)
    M2:  PUB_NR_PONG      (Dummy Reply to EXT_NR_PING)
}

```

Process executed for every Think Step of M1:

- Perform one [calculation](#) over the net i.e. calculate neuron activations and output signals of every neuron in the net.
- For every extern neuron of M1:
 - If no command is currently pending (= no reply is being awaited) then:
 - if timeout of last EXT_NR_REQ_CHNG_ACT_OUT command occurred then:
 - M1: **EXT_NR_REQ_ACT_OUT** (Request activation and output of public counterpart neuron)
 - else
 - M1: **EXT_NR_REQ_CHNG_ACT_OUT** (Request activation and output change of public counterpart neuron)
 - If PUB_NR_REQ_CHNG_INP has been received and the input signal sum of the extern neuron has changed compared to the last transmitted value then:
 - M1: **EXT_NR_INP** (Transmit the input signal sum of the extern neuron to its public counterpart neuron)

- For every public neuron of M1:
 - If no command is currently pending (= no reply is being awaited) then:
 - if timeout of last PUB_NR_REQ_CHNG_INP command occurred then:
 - M1: **PUB_NR_REQ_INP** (Request input sum of extern counterpart neuron)
 - else
 - M1: **PUB_NR_REQ_CHNG_INP** (Request input sum change of public counterpart neuron)
 - If EXT_NR_REQ_CHNG_ACT_OUT has been received and the activation or the output signal of the public neuron has changed compared to the last transmitted value then:
 - M1: **PUB_NR_ACT_OUT** (Transmit the activation and the output signal of the public neuron to its extern counterpart neuron)

Note1:

In MemBrain all commands except for the change requests EXT_NR_REQ_CHNG_ACT_OUT and PUB_NR_REQ_CHNG_INP are answered to immediately. If one of the named two commands is received the reply may be delayed until the requested signals change with respect to their last transmitted values. Nevertheless if the values already have changed when the commands are received then the reply is generated immediately to prevent other devices from unnecessarily waiting.

Transmission data package optimization:

As already denoted [here](#) MemBrain by default operates with the so called 'Nagle Algorithm' being deactivated. This is because MemBrain does its own optimization with respect to data package size:

When a Think Step is performed then all outgoing data frames are streamed into a send buffer (FIFO) in order to build larger frames to be transmitted over the actual TCP connection. This buffer is flushed (handed over to the TCP socket in order to be transmitted) in either one of the following cases:

- All outgoing data of one think step is collected
- The FIFO gets fully filled (flush to prevent overflow)
- At least one reply to a request command has been placed in the FIFO and all pending incoming [link layer frames](#) have been processed.

As a result of this MemBrain normally bundles as much link layer frames into one TCP package as possible. Thus the receive buffer of a MemBrain compatible device should be large enough to capture one link layer frame for every one

of its published or invoked external neurons in order to achieve optimal performance.

Nevertheless all currently defined [commands](#) are significantly smaller in size than the maximum net link layer payload size (100 bytes, see [here](#) for details). Thus if the target has low memory resources the receive buffer can be reduced to a number of 50 bytes per published or extern neuron. Certainly, communication will also work with a smaller receive buffer but performance will significantly decrease because the number of needed TCP packages increases.

Optimal neuron order when communicating:

When MemBrain receives a command for a neuron (public or extern) it has to search for the corresponding neuron the command is dedicated to according to the [addresses framing](#). MemBrain is optimized to search for neuron addresses in the same order as the neurons are delivered when requested by the command [SVR_REQ_PN_ALL](#). This correlates to increasing neuron index delivered by the reply [SVR_PN](#).

Thus, when an application is requesting data for many neurons it should issue the requests in this order as this approach results in the best performance. Also MemBrain uses this order when communicating, so a device should

implement a search algorithm that is optimized to this: When receiving an [addressed command](#) MemBrain first checks if the command belongs to the same neuron as the last received command belonged to. If this is not the case MemBrain starts to search for the neuron in direction of increasing index. If the maximum

index is reached MemBrain starts searching at index 0 up to the position where it originally started the search.

Feedback and Contact

If you have bug findings, questions, suggestions for new releases or any other feedback with regard to MemBrain then please feel free to visit the MemBrain homepage at

www.membrain-nn.de

or contact me directly at

thomas.jetter@membrain-nn.de

Any feedback is highly appreciated, whether in english or in german language.

Thanks in advance to all who help improving MemBrain this way!

Used Licenses

This section provides legal information about licenses for software which is used by MemBrain.

1) KissFFT

For FFT calculations MemBrain uses the library 'KissFFT'.

- a) License: BSD License
- b) License information for KissFFT library and related source code ----- START

```
Copyright (c) 2003-2010, Mark Borgerding
```

```
All rights reserved.
```

```
Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:
```

```
* Redistributions of source code must retain the above copyright notice,
this list of conditions and the following disclaimer.
```

```
* Redistributions in binary form must reproduce the above copyright
notice, this list of conditions and the following disclaimer in the
documentation and/or other materials provided with the distribution.
```

```
* Neither the author nor the names of any contributors may be used to
endorse or promote products derived from this software without specific prior
written permission.
```

```
THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

License information for KissFFT library and related source code ----- END
