UNIVERSITY OF STUTTGART

INSTITUTE FOR PARALLEL AND DISTRIBUTED
HIGH PERFORMANCE SYSTEMS (IPVR)

Applied Computer Science – Image Understanding

SNNS

Stuttgart Neural Network Simulator

Batchman Manual

UNIVERSITY OF STUTTGART

INSTITUTE FOR PARALLEL AND DISTRIBUTED
HIGH PERFORMANCE SYSTEMS (IPVR)

Applied Computer Science – Image Understanding

Günter Mamier, Jens Wieland

SNNS

Stuttgart Neural Network Simulator

Batchman User Manual

Chapter 15

Batchman

This chapter describes `batchman`, a language that enables the user to control SNNS in batch mode.

15.1 Introduction

This newly implemented batch language is to replace the old `SNNSBAT`. Programs which are written in the old `SNNSBAT` language will not be able to run on the newly designed interpreter. The new language supports all functions which are necessary to train and test neural nets. All non-graphical features which are offered by the graphical user interface (`XGUI`) may be accessed with the help of this language as well.

The new batch language was modeled after languages like `AWK`, `Pascal`, `Modula2` and `C`. It is an advantage to have some knowledge in one of the described languages. The language will enable the user to get the desired result without investing a lot of time in learning its syntactical structure. For most operators multiple spellings are possible and variables don't have to be declared before they are used. If an error occurs in the written batch program the user will be informed by a displayed meaningful error message (warning) and the corresponding line number.

15.1.1 Styling Conventions

Here is a description of the style conventions used:

Input which occurs on a Unix command line or which is part of the batch program will be displayed in `typewriter` writing. Such an input should be adopted without any modification.

For example:

```
/Unix> batchman -h
```

This is an instruction which should be entered in the Unix command line, where `/Unix>` is the shell prompt which expects input from the user. Its appearance may change depending on the Unix-system installed. The instruction `batchman -h` starts the interpreter with the `-h` help option which tells the interpreter to display a help message. Every form of input has to be confirmed with Enter (Return). Batch programs or part of batch programs will also be displayed in typewriter writing. Batch programs can be written with a conventional text editor and saved in a file. Commands can also be entered in the interactive mode of the interpreter. If a file is used as a source to enter instructions, the name of the file has to be provided when starting the interpreter. Typewriter writing is also used for wild cards. Those wild cards have to be replaced by real names.

15.1.2 Calling the Batch Interpreter

The Interpreter can be used in an interactive mode or with the help of a file, containing the batch program. When using a file no input from the keyboard is necessary. The interactive mode can be activated by just calling the interpreter:

```
/Unix> batchman
```

which produces:

```
SNNS Batch Interpreter V1.0.  Type batchman -h for help.
No input file specified, reading input from stdin.
batchman>
```

Now the interpreter is ready to accept the user's instructions, which can be entered with the help of the keyboard. Once the input is completed the interpreter can be put to work with `Ctrl-D`. The interpreter can be aborted with `Ctrl-C`. The instructions entered are only invoked after `Ctrl-D` is pressed.

If the user decides to use a file for input the command line option `-f` has to be given together with the name of the interpreter:

```
/Unix> batchman -f myprog.bat
```

Once this is completed, the interpreter starts the program contained in the file myprog.bat and executes its commands.

The standard output is usually the screen but with the command line option `-l` the output can be redirected in a protocol file. The name of the file has to follow the command line option:

```
/Unix> batchman -l logfile
```

Usually the output is redirected in combination with the reading of the program out of a file:

```
/Unix> batchman -f myprog.bat -l logfile
```

The order of the command line options is arbitrary.

More command line options are:

- `-p`: Programs should only be parsed but not executed. This option tells the interpreter to check the correctness of the program without executing the instructions contained in the program. Run time errors can not be detected. Such a run time error could be an invalid SNNS function call.
- `-q`: No messages should be displayed except those caused by the `print()`-function.
- `-s`: No warnings should be displayed.
- `-h`: A help message should be displayed which describes the available command line options.

All following input will be printed without the shell-text.

15.2 Description of the Batch Language

This section explains the general structure of a batch program, the usage of variables of the different data types and usage of the print function. After this an introduction to control structures follows.

15.2.1 Structure of a Batch Program

The structure of a batch program is not predetermined. There is no declaration section for variables in the program. All instructions are specified in the program according to their execution order. Multiple blanks are allowed between instructions. Even no blanks between instructions are possible if the semantics are clear. Single instructions in a line don't have to be completed by a semicolon. In such a case the end of line character (Ctrl-D) is separating two different instructions in two lines. Also key words which have the responsibility of determining the end of a block (`endwhile, endif, endfor, until` and `else`) don't have to be completed by a semicolon. Multiple semicolons are possible between two instructions. However if there are more than two instructions in a line the semicolon is necessary. Comments in the source code of the programs start with a '#' character. Then the rest of the line will be regarded as a comment.
A comment could have the following appearance:

```
#This is a comment
a:=4 #This is another comment
```

The second line begins with an instruction and ends with a comment.

15.2.2 Data Types and Variables

The batch language is able to recognize the following data types:

- Integer numbers
- Floating point numbers
- Boolean type 'TRUE' and 'FALSE'
- Strings

The creation of float numbers is similar to a creation of such numbers in the language C because they both use the exponential representation. Float numbers would be: 0.42, 3e3, or 0.7E-12. The value of 0.7E-12 would be $0.7 * 10^{-12}$ and the value of 3e3 would be $3 * 10^3$

Boolean values are entered as shown above and without any kind of modification.

Strings have to be enclosed by " and can not contain the tabulator character. Strings also have to contain at least one character and can not be longer than one line. Such strings could be:

```
"This is a string"
"This is also a string (0.7E-12)"
```

The following example would yield an error

```
"But this
is not a string"
```

15.2.3 Variables

In order to save values it is possible to use variables in the batch language. A variable is introduced to the interpreter automatically once it is used for the first time. No previous declaration is required. Names of variables must start with a letter or an underscore. Digits, letters or more underscores could follow. Names could be:

```
a, num1, _test, first_net, k17_u, Test_buffer_1
```

The interpreter distinguishes between lower and upper case letters. The type of a variable is not known until a value is assigned to it. The variable has the same type as the assigned value:

```
a = 5
filename := "first.net"
init_flag := TRUE
NET_ERR = 4.7e+11
a := init_flag
```

The assignment of variables is done by using '=' or ':='. The comparison operator is '=='. The variable 'a' belongs to the type integer and changes its type in line 5 to boolean. `Filename` belongs to the type string and `NET_ERR` to the type float.

15.2.4 System Variables

System variables are predefined variables that are set by the program and that are read-only for the user. The following system variables have the same semantics as the displayed variables in the graphical user interface:

SSE	Sum of the squared differences of each output neuron
MSE	SSE divided by the number of training patterns
SSEPU	SSE divided by the number of output neurons of the net
CYCLES	Number of the cycles trained so far.

Additionally there are two more system variables:

PAT	The number of patterns in the current pattern set
EXIT_CODE	The exit status of an execute call

15.2.5 Operators and Expressions

An expression is usually a formula which calculates a value. An expression could be a complex mathematical formula or just a value. Expressions include:

```
3
TRUE
3 + 3
17 - 4 * a + (2 * ln 5) / 0.3
```

The value or the result of an expression can be assigned to a variable. The following operators exist, ordered by priority from top to bottom:

Operator	Function
+, −	Sign for numbers
not, !	Logic negation for boolean numbers
sqrt	Square root
ln	Natural logarithm to the basis e
log	Logarithms to the basis 10
**, ^	Exponential function
*	Multiplication
/	Division
div	Even number division with an even result
mod, %	Result after an even number division
+	Addition
−	Subtraction

Operator	Function
<	smaller than
<=, =<	smaller equal
>	greater than
>=, =>	greater equal
==	equal
<>, !=	not equal
and, &&	logic AND for boolean values
or, ||	logic OR for boolean values

If more than one expression occurs in a line the execution of expressions starts at the left and proceeds towards the right. The order can be changed with parentheses '(' ')'.

The type of an expression is determined at run time and is set with the operator except in the case of integer number division, the modulo operation, the boolean operation and the compare operations.

If two integer values are multilpied, the result will be an integer value. But if an integer and a float value are multilpied, the result will be a float value. If one operator is of type string, then all other operators are transformed into strings. Partial expressions are calculated before the transformation takes place:

```
a := 5 + " plus " + 4 + " is " + ( 8 + 1 )
```

is transformed to the string:

```
5 plus 4 is 9
```

Please note that if the user decides to use operators such as sqrt, ln, log or the exponential operator, no parentheses are required because the operators are not function calls:

Square root:	`sqrt 9`
natural logarithm:	`ln 2`
logarithm to the base of 10:	`log alpha`
Exponential function:	`10 ** 4` oder `a^b`

However parentheses are possible and some times even necessary:

```
sqrt (9 + 16)
ln (2^16)
log (alpha * sqrt tau)
```

15.2.6 The Print Function

So far the user is able to generate expressions and to assign a value to a variable. In order to display values, the print function is used. The print function is a real function call of the batch interpreter and displays all values on the standard output if no input file is declared. Otherwise all output is redirected into a file. The print function can be called with multiple arguments. If the function is called without any arguments a new line will be produced. All print commands are automatically terminated with a newline.

Instruction:	generates the output:
`print(5)`	`5`
`print(3*4)`	`12`
`print("This is a text")`	`This is a text`
`print("This is a text and values:",1,2,3)`	`This is a text and values:123`
`print("Or: ",1," ",2," ",3)`	`Or: 1 2 3`
`print(ln (2^16))`	`11.0904`
`print(FALSE)`	`FALSE`
`print(25e-2)`	`0.25`

If a variable, which has not been assigned a value yet, is tried to be printed, the print function will display $< >$ `undef` instead of a value.

15.2.7 Control Structures

Control structures are a characteristic of a programming language. Such structures make it possible to repeat one or multiple instructions depending on a condition or a value. `BLOCK` has to be replaced by a sequence of instructions. `ASSIGNMENT` has to be replaced by an assignment operation and `EXPRESSION` by an expression. It is also possible to branch within a program with the help of such control structures:

```
if EXPRESSION then BLOCK endif
if EXPRESSION then BLOCK else BLOCK endif
for ASSIGNMENT to EXPRESSION do BLOCK endfor
while EXPRESSION do BLOCK endwhile
repeat BLOCK until EXPRESSION
```

The If Instruction

There are two variants to the `if` instruction. The `first` variant is:

```
If EXPRESSION then BLOCK endif
```

The block is executed only if the expression has the boolean value TRUE.

`EXPRESSIONS` can be replaced by any complex expression if it delivers a boolean value:

```
if sqrt (9)-5<0 and TRUE<>FALSE then print("hello world") endif
```

produces:

```
hello world
```

Please note that the logic operator ‘and’ is the operator last executed due to its lowest priority. If there is confusion about the execution order, it is recommended to use brackets to make sure the desired result will be achieved.

The `second` variant of the `if` operator uses a second block which will be executed as an alternative to the first one. The structure of the second `if` variant looks like this:

```
if EXPRESSION then BLOCK1 else BLOCK2 endif
```

The first BLOCK, here described as BLOCK1, will be executed only if the resulting value of EXPRESSION is 'TRUE'. If EXPRESSION delivers 'FALSE', BLOCK2 will be executed.

The For Instruction

The `for` instruction is a control structure to repeat a block, a fixed number of times. The most general appearance is:

for ASSIGNMENT to EXPRESSION do BLOCK endfor

A counter for the `for` repetitions of the block is needed. This is a variable which counts the loop iterations. The value is increased by one if an loop iteration is completed. If the value of the counter is larger then the value of the `EXPRESSIONS`, the `BLOCK` won't be executed anymore. If the value is already larger at the beginning, the instructions contained in the block are not executed at all. The counter is a simple variable. A `for` instruction could look like this:

for i := 2 to 5 do print (" here we are: ",i) endfor

produces:

```
here we are:  2
here we are:  3
here we are:  4
here we are:  5
```

It is possible to control the repetitions of a block by assigning a value to the counter or by using the `continue`, `break` instructions. The instruction `break` leaves the cycle immediately while `continue` increases the counter by one and performs another repetition of the block. One example could be:

```
for counter := 1 to 200 do
a := a * counter
c := c + 1
if test == TRUE then break endif
endfor
```

In this example the boolean variable test is used to abort the repetitions of the block early.

While and Repeat Instructions

The `while` and the `repeat` instructions differ from a `for` instruction because they don't have a count variable and execute their commands only while a condition is met (while) or until a condition is met (repeat). The condition is an expression which delivers a boolean value. The formats of the `while` and the `repeat` instructions are:

```
while EXPRESSION do BLOCK endwhile
repeat BLOCK until EXPRESSION
```

The user has to make sure that the cycle terminates at one point. This can be achieved by making sure that the EXPRESSION delivers once the value 'TRUE' in case of the `repeat` instruction or 'FALSE' in case of the `while` instruction. The `for` example from the previous section is equivalent to:

```
i := 2
while i <= 5 do
print ( "here we are:  ",i)
i := i + 1 endwhile
```

or to:

```
i := 2
repeat
print ( "here we are:  ",i)
i := i + 1
until i > 5
```

The main difference between `repeat` and `while` is that repeat guarantees that the BLOCK is executed at least once. The `break` and the `continue` instructions may also be used within the BLOCK.

15.3 SNNS Function Calls

The SNNS function calls control the SNNS kernel. They are available as function calls in `batchman`. The function calls can be divided into four groups:

- Functions which are setting SNNS parameters :
  - setInitFunc()
  - setLearnFunc()
  - setUpdateFunc()
  - setPruningFunc()
  - setSubPattern()
  - setShuffle()
  - setSubShuffle()
- Functions which refer to neural nets :
  - loadNet()
  - saveNet()
  - saveResult()
  - initNet()
  - trainNet()
  - testNet()

- Functions which refer to patterns :
  - loadPattern()
  - setPattern()
  - delPattern()
- Special functions :
  - pruneNet()
  - execute()
  - print()
  - exit()
  - setseed()

The format of such calls is:

function_name (parameter1, parameter2...)

No parameters, one parameter, or multiple parameters can be placed after the function name. Unspecified values take on a default value. Note, however, that if the third value is to be modified, the first two values have to be provided with the function call as well. The parameters have the same order as in the graphical user interface.

Now a description of the function calls of the first group follows.

15.3.1 Function Calls To Set SNNS Parameters

The following functions calls to set SNNS parameters are available:

`setInitFunc()`	Selects the initialization function and its parameters
`setLearnFunc()`	Selects the learning function and its parameters
`setUpdateFunc()`	Selects the update function and its parameters
`setPruningFunc()`	Selects the pruning function and its parameters
`setSubPattern()`	Defines the subpattern shifting scheme
`setShuffle()`	Change the shuffle modus
`setSubShuffle()`	Change the subpattern shuffle modus

The format and the usage of the function calls will be discussed now. It is an enormous help to be familiar with the graphical user interface of the SNNS especially with the chapters "Parameters of the learning functions", "Update functions", "Initialization functions", "Handling patterns with SNNS", and "Pruning algorithms".

setInitFunc

This function call selects the function with which the net is initialized. The format is:

```
setInitFunc (function name, parameter...)
```

where `function name` is the initialization function and has to be selected out of:

ART1_Weights	DLVQ_Weights	Random_Weights_Perc
ART2_Weights	Hebb	Randomize_Weights
ARTMAP_Weights	Hebb_Fixed_Act	RBF_Weights
CC_Weights	JE_Weights	RBF_Weights_Kohonen
ClippHebb	Kohonen_Rand_Pat	RBF_Weights_Redo
CPN_Weights_v3.2	Kohonen_Weights_v3.2	RCC_Weights
CPN_Weights_v3.3	Kohonen_Const	RM_Random_Weights
CPN_Rand_Pat	PseudoInv	

It has to be provided by the user and the name has to be exactly as printed above. The function name has to be embraced by "".

After the name of the initialization function is provided the user can enter the parameters which influence the initialization process. If no parameters have been entered default values will be selected. The selected parameters have to be of type float. Function calls could look like this:

```
setInitFunc ("Randomize_Weights")
setInitFunc("Randomize_Weights", 1.0, -1.0)
```

where the first call selects the `Randomize_Weights` function with default parameters. The second call uses the `Randomize_Weights` function and sets two parameters. The batch interpreter displays:

```
Init function is now Randomize_Weights
Parameters are:  1.0 -1.0
```

setLearnFunc

The function call `setLearnFunc` is very similar to the `setinitFunc` call. `setLearnFunc` selects the learning function which will be used in the training process of the neural net. The format is:

```
setLearnFunc (function name, parameters....)
```

where function name is the name of the desired learning algorithm. This name is mandatory and has to match one of the following strings:

ART1	Dynamic_LVQ	RadialBasisLearning
ART2	Hebbian	RBF-DDA
ARTMAP	JE_BP	RCC
BackPercolation	JE_BP_Momentum	RM_delta
BackpropBatch	JE_Quickprop	Rprop
BackpropMomentum	JE_Rprop	Sim_Ann_SS
BackpropWeightDecay	Kohonen	Sim_Ann_WTA
BPTT	Monte-Carlo	Sim_Ann_WWTA
BBPTT	PruningFeedForward	Std_Backpropagation
CC	QPTT	TimeDelayBackprop
Counterpropagation	Quickprop	

After the name of the learning algorithm is provided, the user can specify some parameters. The interpreter is using default values if no parameters are selected. The values have to be of the type float. A detailed description can be found in the chapter "Parameter of the learning function". Function calls could look like this:

```
setLearn("Std_Backpropagation")
setLearn( "Std_Backpropagation", 0.1)
```

The first function call selects the learning algorithm and the second one additionally provides the first learning parameter. The batch interpreter displays:

```
Learning function is now:  Std_backpropagation
Parameters are:  0.1
```

setUpdateFunc

This function is selecting the order in which the neurons are visited. The format is:

```
setUpdateFunc (function name, parameters...)
```

where function name is the name of the update function. The name of the update algorithm has to be selected as shown below.

```
Topological_Order     BAM_Order             JE_Special
ART1_Stable           BPTT_Order            Kohonen_Order
ART1_Synchronous      CC_Order              Random_Order
ART2_Stable           CounterPropagation    Random_Permutation
ART2_Synchronous      Dynamic_LVQ           RCC_Order
ARTMAP_Stable         Hopfield_Fixed_Act    Serial_Order
ARTMAP_Synchronous    Hopfield_Synchronous  Synchonous_Order
Auto_Synchronous      JE_Order              TimeDelay_Order
```

After the name is provided several parameters can follow. If no parameters are selected, default values are chosen by the interpreter. The parameters have to be of the type float. The update functions are described in the chapter `Update functions`. A function call could look like this:

```
setUpdateFunc ("Topological_Order")
```

The batch interpreter displays:

```
Update function is now Topological_Order
```

setPruningFunc

This function call is used to select the different pruning algorithms for neural networks. (See chapter `Pruning algorithms`). A function call may look like this:

```
setPruningFunc (function name1, function name2, parameters)
```

where function name1 is the name of the pruning function and has to be selected from:

```
MagPruning               OptimalBrainSurgeon    OptimalBrainDamage
Noncontributing_Units    Skeletonization
```

Function name2 is the name of the subordinated learning function and has to be selected out of:

```
BackpropBatch          Quickprop         BackpropWeightDecay
BackpropMomentum       Rprop             Std_Backpropagation
```

Additionally the parameters described below can be entered. If no parameters are entered default values are used by the interpreter. Those values appear in the graphical user interface in the corresponding widget of the pruning window.

1. Maximum error increase in % (float)
2. Accepted error (float)
3. Recreate last pruned element (boolean)
4. Learn cycles for first training (integer)
5. Learn cycles for retraining (integer)
6. Minimum error to stop (float)
7. Initial value for matrix (float)
8. Input pruning (boolean)
9. Hidden pruning (boolean)

Function calls could look like this:

```
setPruningFunc("OptimalBrainDamage","Std_Backpropagation")
setPruningFunc("MagPruning", "Rprop", 15.0, 3.5, FALSE, 500, 90,
1e6, 1.0)
```

In the first function call the pruning function and the subordinate learning function is selected. In the second function call almost all parameters are specified. Please note that a function call has to be specified without a carriage return. Long function calls have to be specified within one line. The following text is displayed by the batch interpreter:

```
Pruning function is now MagPruning
Subordinate learning function is now Rprop
Parameters are:  15.0 3.5 FALSE 500 90 1.0 1e-6 TRUE TRUE
```

The regular learning function `PruningFeedForward` has to be set with the function call `setLearnFunc()`. This is not necessary if `PruningFeedForward` is already set in the network file.

setSubPattern

The function call `setSubPattern` defines the *Subpattern-Shifting-Scheme* which is described in chapter "Variable size pattern". The definition of the *Subpattern-Shifting-Scheme* has to fit the used pattern file and the architecture of the net. The format of the function call is:

```
setSubPattern(InputSize, InputStep1, OutputSize1, OutputStep1)
```

The first dimension of the subpatterns is described by the first four parameters. The order of the parameters is identical to the order in the graphical user interface (see chapter "Sub Pattern Handling"). All four parameters are needed for one dimension. If a second dimension exists the four parameters of that dimension are given after the four parameters of the first dimension. This applies to all following dimensions. Function calls could look like this:

```
setSubPattern (5, 3, 5, 1)
setSubPattern(5, 3, 5, 1, 5, 3, 5, 1)
```

A one-dimensional subpattern with the InputSize 5, InputStep 3, OutputSize 5, Output-Step 1 is defined by the first call. A two-dimensional subpattern as used in the example network `watch net` is defined by the second function call. The following text is displayed by the batch interpreter:

```
Sub-pattern shifting scheme (re)defined
Parameters are:  5 3 5 1 5 3 5 1
```

The parameters have to be integers.

setShuffle, setSubShuffle

The function calls `setShuffle` and `setSubShuffle` enable the user to work with the shuffle function of the SNNS which selects the next training pattern randomly. The shuffle function can be switched on or off. The format of the function calls is:

```
setShuffle (mode)
setSubShuffle (mode)
```

where the parameter mode is a boolean value. The boolean value TRUE switches the shuffle function on and the boolean value FALSE switches it off. `setShuffe` relates to regular patterns and `setSubShuffle` relates to subpatterns. The function call:

```
setSubShuffle (TRUE)
```

will display:

```
Subpattern shuffling enabled
```

15.3.2 Function Calls Related To Networks

This section describes the second group of function calls which are related to network or network files. The second group of SNNS functions contains the following function calls:

`loadNet()`	Load a net
`saveNet()`	Save a net
`saveResult()`	Save a result file
`initNet()`	initialize a net
`trainNet()`	train a net
`testNet()`	test a net

The function calls `loadNet` and `saveNet` both have the same format:

```
loadNet (file_name)
saveNet (file_name)
```

where file_name is a valid Unix file name enclosed by `" "`. The function `loadNet` loads a net in the simulator kernel and `saveNet` saves a net which is currently located in the simulator kernel. The function call `loadNet` sets the system variable `CYCLES` to zero. This variable contains the number of training cycles used by the simulator to train a net. Examples for such calls could be:

```
loadNet ("encoder.net")
...
saveNet ("encoder.net")
```

The following result can be seen:

```
Net encoder.net loaded
Network file encoder.net written
```

The function call `saveResult` saves a SNNS result file and has the following format:

```
saveResult (file_name, start, end, inclIn, inclOut, file_mode)
```

The first parameter (file_name) is required. The file name has to be a valid Unix file name enclosed by `" "`. All other parameters are optional. Please note that if one specific parameter is to be entered all other parameters before the entered parameter have to be provided also. The parameter `start` selects the first pattern which will be handled and `end` selects the last one. If the user wants to handle all patterns the system variable PAT can be entered here. This system variable contains the number of all patterns. The parameters `inclIn` and `inclOut` decide if the input patterns and the output patterns should be saved in the result file or not. Those parameters contain boolean values. If `inclIn` is TRUE all input patterns will be saved in the result file. If `inclIn` is FALSE the patterns will not be saved. The parameter `inclOut` is identical except for the fact that it relates to output patterns. The last parameter `file_mode` of the type string, decides if a file should be created or if data is just appended to an existing file. The strings "create" and "append" are accepted for file mode. A `saveResult` call could look like this:

```
saveResult ("encoder.res")
saveResult ("encoder.res", 1, PAT, FALSE, TRUE, "create")
```

both will produce this:

```
Result file encoder.res written
```

In the second case the result file encoder.res was written and contains all output patterns.

The function calls `initNet, trainNet, testNet` are related to each other. All functions are called without any parameters:

```
initNet()
trainNet()
testNet()
```

`initNet()` initializes the neural network. After the net has been reset with the function call `setInitFunc`, the system variable `CYCLE` is set to zero. The function call `initNet` is necessary if an untrained net is to be trained for the first time or if the user wants to set a trained net to its untrained state.

```
initNet()
```

produces:

```
Net initialized
```

The function call `trainNet` is training the net exactly one cycle long. After this, the content of the system variables `SSE, MSE, SSEPU` and `Cycles` is updated.

The function call `testNet` is used to display the user the error of the trained net, without actually training it. This call changes the system variables `SSE , MSE, SSEPU` but leaves the net and all its weights unchanged.

Please note that the function call `trainNet` is usually used in combination with a repetition control structure like `for`, `repeat`, or `while`.

15.3.3 Pattern Function Calls

The following function calls relate to patterns:

`loadPattern()`	Loads the pattern file
`setPattern()`	Replaces the current pattern file
`delPattern()`	Deletes the pattern file

The simulator kernel is able to store several pattern files (currently 5). The user can switch between those pattern files with the help of the `setPattern()` call. The function call `delPattern` deletes a pattern file from the simulator kernel. All three mentioned calls have `file_name` as an argument:

```
loadPattern (file_name)
setPattern (file_name)
delPattern (file_name)
```

All three function calls set the value of the system variable `Pat` to the number of patterns of the pattern file used last. The handling of the pattern files is similar to the handling of such files in the graphical user interface. The last loaded pattern file is the current one. The function call `setPattern` (similar to the USE button of the graphical user interface of the SNNS.) selects one of the loaded pattern files as the one currently in use. The call `delPattern` deletes the pattern file currently in use from the kernel. The function calls:

```
loadPattern ("encoder.pat")
loadPattern ("encoder1.pat")
setPattern("encoder.pat")
delPattern("encoder.pat")
```

produce:

```
Patternset encoder.pat loaded; 1 patternset(s) in memory
Patternset encoder1.pat loaded; 2 patternset(s) in memory
Patternset is now encoder.pat
Patternset encoder.pat deleted; 1 patternset(s) in memory
Patternset is now encoder1.pat
```

15.3.4 Special Functions

There are four miscelaneous functions for the use in `batchman`

`pruneNet()`	Starts network pruning
`execute()`	Executes any unix shell comand or program
`exit()`	Quits `batchman`
`setseed()`	Sets a seed for the random number generator

The function call pruneNet() is pruning a net equivalent to the pruning in the graphical user interface. After all functions and parameters are set with the call `setPruningFunc` the `pruneNet()` function call can be executed. No parameters are necessary:

pruneNet()

An interface to the Unix operation system can be created by using the function `execute`. This function call enables the user to start a program at the Unix command line and redirect its output to the batch program. All Unix help programs can be used to make this special function a very powerful tool. The format is:

```
execute (instruction, variable1, variable2.....)
```

where 'instruction' is a Unix instruction or a Unix program. All output, generated by the Unix command has to be separated by blanks and has to be placed in one line. If this is not done automatically please use the Unix commands `AWK` or `grep` to format the output as needed. Those commands are able to produce such a format. The output generated by the program will be assigned, according to the order of the output sequences, to the variables `variable1`, `variable2`.. The data type of the generated output is automatically set to one of the four data types of the batch interpreter. Additionally the exit state of the Unix program is saved in the system variable `EXIT_CODE`. An example for `execute` is:

```
execute ("date", one, two, three, four)
print ("It is ", four, " o'clock")
```

This function call calls the command `date` and reads the output `"Fri May 19 16:28:29 GMT 1995"` in the four above named variables. The variable 'four' contains the time. The batch interpreter produces:

```
It is 16:28:29 o'clock
```

The `execute` call could also be used to determine the available free disk space:

```
execute ("df .| grep dev", dmy, dmy, dmy, freeblocks)
print ("There are ", freeblocks, "Blocks free")
```

In this examples the Unix pipe and the grep command are responsible for reducing the output and placing it into one line. All lines, that contain `dev`, are filtered out. The second line is read by the batch interpreter and all information is assigned to the named variables. The first three fields are assigned to the variable `dmy`. The information about the available blocks will be stored in the variable `freeblocks`. The following output is produced:

```
There are 46102 Blocks free
```

The examples given above should give the user an idea how to handle the `execute` command. It should be pointed out here that `execute` could as well call another batch interpreter which could work on partial solutions of the problem. If the user wants to accomplish such a task the command line option -q of the batch interpreter could be used to suppress output not caused by the print command. This would ease the reading of the output.

```
exit()
```

The last special function call is `exit`. This function call leaves the batch program immediately and terminates the batch interpreter. The parameter used in this function is the exit state, which will be returned to the calling program (usually the Unix shell). If no parameter is used the batch interpreter returns zero. The format is:

```
exit (state)
```

The integer `state` ranges from -128 to +127. If the value is not within this range the value will be mapped into the valid range and an error message displayed. The following example will show the user how this function call could be used:

```
if freeblocks < 1000 then
print ("Not enough disk space")
exit (1)
endif
```

The function `setseed` sets a seed value for the random number generator used by the initialization functions. If `setseed` is not called before initializing a network, subsequent initializiations yield the exact same initial network conditions. Thereby it is possible to make an exact comparison of two training runs with different learning parameters.

```
setseed(seed)
```

`Setseed` may be called with an integer parameter as a seed value. Without a parameter it uses the value returned by the shell command 'date' as seed value.

15.4 Batchman Example Programs

15.4.1 Example 1

A typical program to train a net may look like this:

```
loadNet("encoder.net")
loadPattern("encoder.pat")
setInitFunc("Randomize_Weights", 1.0, -1.0)
initNet()

while SSE > 6.9 and CYCLES < 1000 do
  if CYCLES mod 10 == 0 then
    print ("cycles = ", CYCLES, "  SSE = ", SSE) endif
  trainNet()
endwhile

saveResult("encoder.res", 1, PAT, TRUE, TRUE, "create")
saveNet("encoder.trained.net")

print ("Cycles trained: ", CYCLES)
print ("Training stopped at error: ", SSE)
```

This batch program loads the neural net 'encoder.net' and the corresponding pattern file. Now the net is initialized. A training process continues until the SSE error is smaller or equal to 6.9. The trained net and the result file are saved once the training is completed. The following output is generated by this program:

```
Net encoder.net loaded
Patternset encoder.pat loaded; 1 patternset(s) in memory
Init function is now Randomize_Weights
Net initialised
cycles = 0  SSE = 3.40282e+38
cycles = 10  SSE = 7.68288
cycles = 20  SSE = 7.08139
cycles = 30  SSE = 6.95443
Result file encoder.res written
```

```
Network file encoder.trained.net written
Cycles trained: 40
Training stopped at error: 6.89944
```

15.4.2 Example 2

The following example program reads the output of the network analyzation program `analyze`. The output is transformed into a single line with the help of the program `analyze.gawk`. The net is trained until all patterns are classified correctly:

```
loadNet ("encoder.net")
loadPattern ("encoder.pat")
initNet ()

while(TRUE)
  for i := 1 to 500 do
    trainNet ()
  endfor

  resfile := "test.res"
  saveResult (resfile, 1, PAT, FALSE, TRUE, "create")
  saveNet("enc1.net")

  command := "analyze -s -e WTA -i " + resfile + " | analyze.gawk"
  execute(command, w, r, u, e)
  print("wrong: ",w, "  right: ",r, "  unknown: ",u, "  error: ",e)
  if(right == 100) break
endwhile
```

The following output is generated:

```
Net encoder.net loaded
Patternset encoder.pat loaded; 1 patternset(s) in memory
-> Batchman warning at line 3:
   Init function and params not specified; using defaults
Net initialised
Result file test.res written
Network file enc1.net written
wrong: 87.5  right: 12.5  unknown: 0  error: 7
Result file test.res written
Network file enc1.net written
wrong: 50  right: 50  unknown: 0  error: 3
Result file test.res written
Network file enc1.net written
wrong: 0  right: 100  unknown: 0  error: 0
```

15.4.3 Example 3

The last example program shows how the user can validate the training with a second pattern file. The net is trained with one training pattern file and the error, which is used to determine when training should be stopped, is measured on a second pattern file. Thereby it is possible to estimate if the net is able to classify unknown patterns correctly:

```
loadNet ("test.net")
loadPattern ("validate.pat")
loadPattern ("training.pat")
initNet ()

repeat
  for i := 1 to 20 do
    trainNet ()
  endfor
  saveNet ("test." + CYCLES + "cycles.net")
  setPattern ("validate.pat")
  testNet ()
  valid_error := SSE
  setPattern ("training.pat")
until valid_error < 2.5

saveResult ("test.res")
```

The program trains a net for 20 cycles and saves it under a new name for every iteration of the repeat instruction. Each time the program tests the net with the validation pattern set. This process is repeated until the error of the validation set is smaller than 2.5