APRIL-ANN – A Pattern Recognizer in Lua (with ANNs) –

v0.4.1

ALL AUTHORS

Contents

Chapter 1

Introduction

APRIL-ANN (A Pattern Recognizer In Lua with Artificial Neural Networks) is more than an ANNs toolkit. It is pattern recognizer project.

Simple Lua scripts could be implemented to run ANNs experiments. Some examples are below.

1.1 Inline help

Take note that APRIL-ANN offers an inline help with two basic commands:

```
april_help(...)
april_dir(...)
april_list(...)
```

The `april_help(object)` function takes an object (Lua table, function, userdata, ...) as a parameter and shows the corresponding help via standard output.

The `april_dir(object)` function takes an object as a parameter and shows the corresponding help via standard output. It is the same as `april_help` but a lot less verbose.

The `april_list(table)` function takes a table and shows its content using `pairs` function. It has nothing to do with inline help, but is useful in a lot of circumstances when developing scripts.

Play a little with it, so execute `april_help(ann.components)` and after `april_help(ann.components.base)` and see what happens ;)

If you want to access instance methods documentation, you have two ways:

- Use the operator `..` (that is, `__concat` metamethod) with a class table plus method name string:

```
> april_help(ann.components.base .. "forward")
method  Computes forward step with the given token

description: Computes forward step with the given token

parameters:
                    1 An input token (usually a matrix)
                    2 A boolean indicating if the forward is
                          during_training or not. This information is used by
```

```
                        ann.components.actf objects to apply dropout
                        during training, and to halve the activation during
                        validation (or test). It is [optional], by default
                        is false.

outputs:
                    1 An output token (usually a matrix)
```

- Declare an instance of the class and execute `april_help(obj)`:

```
> c = ann.components.base()
> april_help(c.forward)
method  Computes forward step with the given token

description: Computes forward step with the given token

parameters:
                    1 An input token (usually a matrix)
                    2 A boolean indicating if the forward is
                          during_training or not. This information is used by
                          ann.components.actf objects to apply dropout
                          during training, and to halve the activation during
                          validation (or test). It is [optional], by default
                          is false.

outputs:
                    1 An output token (usually a matrix)
```

1.2 Auto-completion

APRIL-ANN incorporates an adaptation of https://github.com/rrthomas/lua-rlcompleter for Lua 5.2, and for the APRIL-ANN object oriented implementation. It allows to auto-complete pathnames, global names, table fields, and object methods, by using the `<tab>` key.

```
> matrix.<tab><tab>
_NAME               dict               fromString         ...
_VERSION            fromFilename       fromTabFilename    ...
__sliding_window__  fromHEX            join               ...
as                  fromMMap           loadImage          ...
...                 ...                ...
> matrix.fromTab<tab>
> matrix.fromTabFilename<tab><tab>
fromTabFilename
> matrix.fromTabFilename
```

1.3 Serialization

Almost any object can be serialized to a disk file, string or a stream using the function `util.serialize()`. Similarly, it can be deserialized using `util.deserialize()` function.

1.4 XOR problem

The code described here is at the repo path `EXAMPLES/xor.lua`. First, we need to create an ANN component object which will be trained:

```
thenet  = ann.mlp.all_all.generate("2 inputs 2 logistic 1 logistic")
```

The object `thenet` is a Multilayer Perceptron (MLP) with 2 inputs, a hidden layer with 2 neurons with logistic activation function, and 1 output neuron with logistic activation function. Some activation functions are available: logistic, tanh, linear, softmax, log_logistic, sin, softsign, softplus, ... (see `april_help(ann.components.actf)`).

Now, in order to do easy and fast development of scripts, a trainer helper wrapper can be used:

```
bunch_size=4
trainer = trainable.supervised_trainer(thenet, ann.loss.mse(1), bunch_size)
```

The trainer needs the ANN component, the loss function, and the bunch_size. Bunch size is the same as mini-batch size, it is used to train several patterns at the same time, increasing the speed of the experiment. Values between 32 and 64 are tipically used, but in this example onlt 4 is possible, so the XOR problem is composed by 4 patterns.

The next step is to build the component and randomize its weights:

```
trainer:build()
trainer:randomize_weights{
  random     =  random(1234),
  inf        = -0.1,
  sup        =  0.1 }
```

The weights will be initialized uniformly in the range [inf, sup], using the given `random` object with 1234 as random seed. It is also possible to indicate if you want to initialize weights.

The components has several learning parameters which needs to be configured:

```
trainer:set_option("learning_rate", 1.0)
trainer:set_option("momentum",      0.5)
trainer:set_layerwise_option("w.*", "weight_decay",  1e-05)
```

Data to train the ANN is defined using `matrix` and `dataset` objects. It is possible to build XOR problem on a `matrix` and use it as training `datasets`:

```
m_xor = matrix.fromString[[
    4 3
    ascii
    0 0 0
    0 1 1
    1 0 1
    1 1 0
]]
ds_input  = dataset.matrix(m_xor, {patternSize={1,2}})
ds_output = dataset.matrix(m_xor, {offset={0,2}, patternSize={1,1}})
```

The variable `m_xor` is a matrix object, loaded from the given string. `ds_input` is a `dataset.matrix` object, which traverses the matrix by rows, computing a sliding window of `patternSize={1,2}`. The desired output of the ANN is another `dataset.matrix`, but in this case computing the sliding window with size (1,1) and skipping the first two columns `offset={0,2}`.

Finally, we need to train the ANN:

```
for i=1,10000 do
  local error = trainer:train_dataset{ input_dataset  = ds_input,
                                       output_dataset = ds_output }
  print(i, error)
end
```

This code trains the ANN for 10,000 epochs, feeding the ANN with `input_dataset` and using as desired output the given `output_dataset`. Patterns are grouped at mini-batches of size 4 (bunch_size), and each training epoch is the training with the full dataset.

This simple example gives you some insight about how to use APRIL-ANN toolkit, but it is not useful in a bit more complicated problems. Next section will explain DIGITS problem, which trains an ANN to classify handwritten digits.

1.5 DIGITS task

The task aborded at this section is classification of handwritten digits. The code is at `EXAMPLES/digits.lua` , and could be executed following this command: `april-ann digits.lua`. This task uses as data a large PNG image with handwritten digits ordered by columns and rows. Each columns corresponds to each digit class (from 0 to 9), and each row contains 10 examples (one for each class). There are 1000 patterns (100 for each clasS). So, first the image is loaded using this code, and converted to a matrix where 0 represents white color and 1 represents black color:

```
digits_image = ImageIO.read(string.get_path(arg[0])..."digits.png")
m1           = digits_image:to_grayscale():invert_colors():matrix()
```

This code uses `ImageIO.read` function to load the PNG image (you need to compile libpng package), and uses `string.get_path` function in order to find where the file is located. The image is converted to grayscale, colors are inverted to be 0=white and 1=black, and finally the corresponding matrix of this image is generated.

Second, the training input and output dataset are generated following this code:

```
-- TRAINING --
train_input  = dataset.matrix(m1,
                              {
                                patternSize = {16,16},
                                offset      = {0,0},
                                numSteps    = {80,10},
                                stepSize    = {16,16},
                                orderStep   = {1,0}
                              })
-- a simple matrix for the desired output
m2 = matrix(10,{1,0,0,0,0,0,0,0,0,0})
-- a circular dataset which advances with step -1
train_output = dataset.matrix(m2,
                              {
```

```
                                  patternSize = {10},
                                  offset      = {0},
                                  numSteps    = {800},
                                  stepSize    = {-1},
                                  circular    = {true}
                                })
```

This is a more complicated example of how to create datasets from matrices. The variable `train_input` is a `dataset.matrix` generated by a sliding-window of size 16x16 (the size of one digit), which moves in steps of 16x16 (first 16 in columns, and when arrive to the end it moves 16 in rows and returns to column 0). The number of patterns (`numSteps`) is 80 by rows and 10 by columns. The output dataset needs an special matrix which contains only one 1 and 9 zeroes, so the 1 on each pattern will correspond to its class. The `dataset.matrix` in this case slides backwards (`stepSize={-1}`), so the 1 moves forward, and is circular (window positions out of the matrix take the values of the opposite matrix positions). It has 800 patterns (80x10).

For validation datasets the script is coded similarly:

```
-- VALIDATION --
val_input = dataset.matrix(m1,
                           {
                             patternSize = {16,16},
                             offset      = {1280,0},
                             numSteps    = {20,10},
                             stepSize    = {16,16},
                             orderStep   = {1,0}
                           })
val_output   = dataset.matrix(m2,
                              {
                                patternSize = {10},
                                offset      = {0},
                                numSteps    = {200},
                                stepSize    = {-1},
                                circular    = {true}
                              })
```

However, in this case the `val_input` dataset needs the option parameter `offset` to not be 0, because validation patterns are the 200 last patterns (it begins at image row position 1280). The first 800 digits are used for training.

The MLP is generated following same steps as for XOR, but in this case the topology description string uses tanh for activation of hidden layer, and log_softmax for activation of output layer. In this case the `use_fanin` and `use_fanout` flags are set to true, and the error function is `multi_class_cross_entropy`, which is a version of cross-entropy error function, but mathematically simplified for log_softmax as output activation functions (if you try other output you must use `mse`). The two-class version of cross-entropy (`ann.loss.cross_entropy`) is simplified to be used with log_logistic outputs:

```
bunch_size = 64
thenet = ann.mlp.all_all.generate("256 inputs 128 tanh 10 log_softmax")
trainer = trainable.supervised_trainer(thenet,
                                       ann.loss.multi_class_cross_entropy(),
                                       bunch_size)
trainer:build()
trainer:randomize_weights{
```

```
  random      = random(52324),
  use_fanin   = true,
  use_fanout  = true,
  inf         = -1,
  sup         =  1,
}
trainer:set_option("learning_rate", 0.01)
trainer:set_option("momentum",      0.01)
trainer:set_layerwise_option("w.*", "weight_decay",  1e-05)
```

For training, it is needed to declare a table which contains the pair input/output datasets and some specific parameters (i.e. shuffle random object to train each epoch with a different permutation of patterns):

```
training_data = {
  input_dataset  = train_input,
  output_dataset = train_output,
  shuffle        = random(25234),
}

validation_data = {
  input_dataset  = val_input,
  output_dataset = val_output,
}
```

The final snippet code train the MLP using holdout-validation, following a stopping criterion which depends on the relative value between current_epoch/best_validation_epoch: when this proportion is greater than 2 the training is stopped (that is, MLP training will stop at 200 epochs if the last best validation epoch is at epoch 100; MLP training will stop at 400 epochs if the last best validation epoch is at epoch 200). Stopping criterion is selected using function helper `trainable.stopping_criteria.make_max_epochs_wo_imp_relative`, and the MLP is trained using the class `trainable.train_holdout_validation`. This last class receives a table which fields are self-explanatory, and follows a holdout-validation algorithm in its `execute` method, and after each epoch `get_state_string` method is used for output facilities.

```
print("# Epoch  Training  Validation  BestEpoch  BestValidation")
stopping_criterion =
  trainable.stopping_criteria.make_max_epochs_wo_imp_relative(2)
train_func = trainable.train_holdout_validation{
    min_epochs         = 4,
    max_epochs         = 1000,
    stopping_criterion = stopping_criterion,
}
clock = util.stopwatch()
clock:go()
epoch_function = function()
                   local tr_loss = trainer:train_dataset(training_data)
                   local va_loss = trainer:validate_dataset(validation_data)
                   return trainer,tr_loss,va_loss
                 end
while train_func:execute(epoch_function) do
  print(train_func:get_state_string())
end
clock:stop()
cpu,wall   = clock:read()
```

```
num_epochs = result.last_epoch
printf("# Wall total time: %.3f    per epoch: %.3f\n", wall, wall/num_epochs)
printf("# CPU  total time: %.3f    per epoch: %.3f\n", cpu, cpu/num_epochs)
printf("# Validation error: %f", result.best_val_error)
```

1.6 Final remarks

This introduction explains you the basic steps to write and execute scripts for pattern recognition using ANNs and the toolkit APRIL-ANN. Please, feel free to use this scripts as initial template for yours ;)

APRIL-ANN has a lot of interesting features. The following list show the most important features, which are detailed in the following sections of this documentation:

- Multidimensional `matrix` library. It allows to perform efficient mathematical operations in Lua.
- Abstract token definition. A token represents *anything*, and is used in several parts of the toolkit for information interchange: `matrix` instances can be wrapperd into a `tokens.matrix` instance, and they are interchangable in ANN components.
- Dataset abstraction. It has the ability to build powerful sliding windows over matrices. At the same time, it is possible to filter datasets producing new datasets on-the-fly. Two abstraction exists: `dataset`, and `dataset.token`.
- Artificial neural networks. Different packages are implemented to perform efficient training of ANNs. Three main concepts: ANN component, loss function and optimization algorithm.
- Trainable package. This package knows all the ANNs stuff, and is a good start point to work with ANNs. Implements a lot of useful code for intrspection, training and testing.
- Random package. The generation of pseudo-random numbers is in this package.
- Automatic differentiation. For more advanced machine learning, an experimental library for automatic differentiation has been added. It allows to specify totally more general models than ANNs abstraction, but with an important loss in efficiency. However, it is useful to do cool things for research with a little implementation effort, before implement them in ANNs.
- Matlab package. It allows to load (not save) matrices and data in **MAT** format. It stills in experimental phase, but the most important things are available.
- Statistics package. Look here for some statistics standard techniques. PCA, running mean and variance computation, pearson correlation, ...
- Complex numbers. In experimental phase, APRIL-ANN allows to work with complex numbers, and complex matrices.
- Util package. It contains a lot of utilities for Lua script development.
- GZIO package. This is the binding of libZ for load/save of compressed files.
- Image and ImageIO packages. The class Image allows to work with color or gray images. The package ImageIO implements useful functions for generic read/write of images, depending in their extension.

Chapter 2

`matrix` package

This documentation is structured as follows:

- Introduction
- Matrix basic methods like `dim()`, `get()`, `rewrap()`, `select()`, ...
- Data initialization methods like `fill()`, `zeros()`, `linspace()`, ...
- Serialization as `toString()`, `toFilename()`, ...
- Low level access as `size()`, `stride()`, `offset()`, `data()`, ...
- Sliding window
- Math operations using stndard operators.
- BLAS API as `axpy()`, `gemv()`, `gemm()`, ...
- LAPACK API as `svd()`, `inv()`, ...
- Component-wise operations as `tan()`, `tanh()`, ...
- Matrix level operations as `min()`, `sum()`, `eq()`, ...
- Indexing and sorting as `index()`, `indexed_fill()`, `order()`, ...
- Matrix functions as `repmat()`, `triu()`, ...
- Matrix extensions as `real_fftwh()`, `iterate()`, `convolution()`.
- Sparse matrix
- Other matrix flavours
- Matrix dictionaries

2.1 Introduction

Package `matrix` could be loaded via the standalone binary, or in Lua with `require("aprilann.matrix")`.

A matrix is a multidimensional data container, by default *float* data. It is similar to the concept of **tensor**, as defined in libraries like Torch. This notion of tensor is not to be confused with *tensors* in physics and engineering, known as *tensor fields*.

The data would be stored following `row_major` order by default, but different methods can change the stride of the dimensions. Additionally a `matrix` can be **transposed** or not, being the transposition a symbolical state, sharing the same data reference as the non-transposed `matrix`.

From Lua, a matrix is declared using one of the available constructors:

```
> -- row major constructor
> m1 = matrix(3,3) -- this is a uninitialitzed 3x3 matrix of floats
```

```
> -- It is also possible to receive a table with data (in row-major order)
> m2 = matrix(3,3, {1, 2, 3, 4, 5, 6, 7, 8, 9})
> print(m2)
1 2 3
4 5 6
7 8 9
# Matrix of size [3,3] in row_major [0x23d5b20 data= 0x23d5e90]
```

Observe that `print` function shows the same for `m2` and `m3`, but internally the data is in different order. The pretty print of a matrix shows the data and a commented line with the size of the matrix and two memory pointers values, the first is the pointer to the C++ object related with the given matrix, and the second is a pointer to the C++ data object where values are stored (memory block, explained below).

The matrix and its data is separated to allow the declaration of sub-matrices:

```
> m4 = m2:slice({2,1},{1,3})
> print(m4)
4 5 6
# Matrix of size [1,3] in row_major [0x2218a90 data= 0x23d5e90]
```

In this case, the matrix `m4` is a slice which begins at position `{2,1}` of matrix `m2`, and has sizes `{1,3}` in each dimension. Note that the matrix pointer is different, but the data pointer is the same as for `m2` (any change to `m4` will be reflected at `m2`.)

Besides, it is possible to do a sub-matrix cloning the data (deep copy) if you add to `slice` method a new boolean argument with `true` value:

```
> m5 = m2:slice({2,1},{1,3}, true)
> print(m5)
4 5 6
# Matrix of size [1,3] in row_major [0x220c6f0 data= 0x2250d60]
```

A shortcut for slicing is to use the operator `m[...]`, which has a meaning similar to Matlab or Octave. It is overloaded and it is parsed into a `slice` or `select` method call, so, it is **slower** than using directly the `slice` or `select` methods, but it is easier to understand and use. This operator receives as argument a number or a table.

- `m[n]` In case of given a number, the operator is equivalent to a `m:select(1,n)` in case the matrix has more than one dimension, and returns a `matrix` object with one less dimension. In case of uni-dimensional matrix, this call is equivalent to `m:get(n)` and returns a Lua number value.
- `m[{s1,s2,...}]` In case of given a table, the should be the same length of the matrix number of dimensions. In every dimension position, three possible values could be given:
  - A number, indicating an exact dimension position: `m = m2[{2,3}]`
  - A table, with start and end positions in the dimension: `m = m2[{2, {1,3}}]`
  - A string, with start and end positions in the dimension: `m = m2[{2, '1:3'}]`

The following examples are equivalent:

```
> = m2:slice({2,1},{1,3})
 4           5           6
# Matrix of size [1,3] in row_major [0xf44470 data= 0xdeae90]
```

```
> = m2[{2, {1,3}]}
 4           5           6
# Matrix of size [1,3] in row_major [0xf3c8d0 data= 0xdeae90]
> = m2[{2, '1:3'}]
 4           5           6
# Matrix of size [1,3] in row_major [0xe32dc0 data= 0xdeae90]
```

It is possible to use the string shortcut to indicate a whole dimension, or only start/end positions:

```
> = m2[{':', '2:3'}]
 2           3
 5           6
 8           9
# Matrix of size [3,2] in row_major [0xef1260 data= 0xdeae90]
> = m2[{'2:', '2:3'}]
 5           6
 8           9
# Matrix of size [2,2] in row_major [0xe08f50 data= 0xdeae90]
> = m2[{':2', '2:3'}]
 2           3
 5           6
# Matrix of size [2,2] in row_major [0xf04290 data= 0xdeae90]
```

A different shortcut exists for assignation operator, using `newindex` metamethod. The matrix will be indexed using `[]` operator or `[{}]` operator:

```
> -- assigns a 0 to all the row values at columns 2:3
> m2[{ ':', '2:3' }] = 0
> -- assigns another matrix all the row values at columns 2:3
> m2[{ ':', '2:3' }] = matrix(m2:dim(1),2):linspace()
> -- assigns 5 to all the values at position 2 in dimension 1
> m2[2] = 5
> -- assigns 10 to all values less than 3
> m2[m2:lt(3)] = 10
```

2.2 Memory block

Every `matrix` has an underlying block of memory which is reinterpreted with the `matrix` shape (dimension sizes, and offset). It is possible to build a memory block in Lua by using any of the `mathcore.block` table constructors. Currently, `mathcore.block.float`, `mathcore.block.double` and `mathcore.block.int32` are available.

```
> b = mathcore.block.float{1,2,3,4,5} -- initialized to a table
> print(b)
 1            2            3            4            ...  5
# Float block size 5 [data= 0x1ab98e0]
> b = mathcore.block.float(20) -- not initialized
```

Every memory block has three basic methods:

- `n = b:size()` returns the number of elements in the memory block.

- `v = b:raw_get(p)` returns the value at the given `p` position. **Note** that the position is **0-indexed** as in C, because this is a low-level C++ method.
- `b:raw_set(p, v)` sets vas the value at the given `p` position. **Note** that the position is 0-indexed as in C.

Like in `matrix` objects, it is possible to use `operator[]`:

- `v = b[p]` is equivalent to `v = b:raw_get(p)`.
- `b[p] = v` is equivalent to `b:raw_set(p, v)`.

Remember that memory blocks are indexed from 0, like in C, because they are a wrapper around a C pointer.

2.3 Hints

NOTICE: Almost all matrix methods returns the caller matrix (when it is possible), allowing to chain transformation sequences.

NOTICE: In Lua the arrays start at 1 instead of 0, so, in the `matrix` methods the dimensions start at 1.

2.4 Constructor

There are different valid `matrix` constructors.

- `m = matrix(d1, d2, ..., dn)` a list of dimension sizes.
- `m = matrix(d1, d2, ..., dn, table)` a list of dimension sizes besides an array table with all the `matrix` elements in row-major order.
- `m = matrix(d1, d2, ..., dn, block)` a list of dimension sizes plus a float memory block (`mathcore.block.float`).
- `m = matrix{ e1, e2, ..., en }` creates a one-dimensional `matrix` with the given data elements.
- `m = matrix(block)` creates a one-dimensional `matrix` with the given float memory block as data.

2.5 MMapped matrix

`m = matrix.MMapped(...)`

It is possible to force the declaration of `matrix` memory as a mmapped anonymous file. This function receives exactly same arguments as the default `matrix` constructor.

```
> -- the following matrix will be allocated as mmapped memory
> -- shared when forking the process
> m = matrix.MMapped(2,2):linear()
> print(m)
 0             1
 2             3
# Matrix of size [2,2] stride [2,1] ref [0x14c1f50 data= 0x14c2020]
```

Another way is to serialize a matrix in MMap format (see serialization section).

2.6 Basic matrix methods

2.6.1 dim

```
[table | number] = m:dim([number])
```

It returns the size of matrix dimensions. Without arguments, it returns a Lua table with the sizes. If an argument is given, it returns the size of the given dimension.

```
> a = matrix(4,3,2)
> = a:dim()
table: 0x23a4780
> = table.concat(a:dim(), " ")
4 3 2
> = a:dim(1), a:dim(2), a:dim(3)
4   3   2
```

2.6.2 get

```
number = m:get(p1, p2, ...)
```

This method returns the value of a given matrix position.

```
> a = matrix(3,4,{1,2,3,4, 5,6,7,8, 10,11,12,13})
> = a:get(1,1)
1
> = a:get(2,3)
7
```

2.6.3 set

```
matrix = m:set(p1, p2, ..., value)
```

This method sets the value of a matrix position, and returns the caller matrix, allowing a sequence of sets.

```
> a = matrix(3,4,{1,2,3,4, 5,6,7,8, 10,11,12,13})
> a:set(2,3, 10000)
> a:set(2,4, 500):set(4,1, 200)
> = a
1 2 3 4
5 6 10000 500
200 11 12 13
# Matrix of size [3,4] in row_major [0x27093d0 data= 0x2709960]
```

2.6.4 clone

```
matrix = m:clone()
```

It allows to clone matrices (deep copy). If the caller `matrix` was transposed, the resulting clone will contain the data in a transposed shape, but with its stride in row major order.

```
> a = matrix(2,3,{1,2,3, 4,5,6}) -- row major matrix
> b = a:clone()                  -- clone (or deep copy) of a
> c = a:transpose()              -- clone of a with different order
> c = c:clone()                  -- clone of a in row major order
```

2.6.5 copy_from_table

```
matrix = m:copy_from_table(table)
```

This method copies the data in the given table into the caller `matrix`, traversing the `matrix` in `row_major` order, as in `matrix` constructor. The table must fit in `matrix` size. The caller `matrix` is returned.

```
> a = matrix(2,3)
> a:copy_from_table({1,2,3, 4,5,6})
```

2.6.6 map

```
matrix = m:map(m1, m2, ..., function)
```

Maps the matrix values by a given list of matrices and a Lua map function. The Lua function will be called for every possible matrix position. The Lua function receives the caller matrix value at the given position, the value of the second matrix, the value of the third matrix, and so on. The Lua function returns `nil`, or **only** one value which will be assigned to the caller matrix **in-place**. All the matrices must have the same dimension sizes. The number of given matrices could be >= 0.

```
> m  = matrix(2,2):linear()
> m2 = matrix(2,2):linear(10)
> m3 = matrix(2,2):linear(100)
> = m
 0           1
 2           3
# Matrix of size [2,2] in row_major [0x1f12050 data= 0x1f0f6a0]
> = m2
 10          11
 12          13
# Matrix of size [2,2] in row_major [0x1f11cc0 data= 0x1f12110]
> = m3
 100         101
 102         103
# Matrix of size [2,2] in row_major [0x1f12740 data= 0x1f11e00]
> m:map(m2,m3,function(x,y,z) return x+y+z end)
> = m
 110         113
 116         119
# Matrix of size [2,2] in row_major [0x1f12050 data= 0x1f0f6a0]
```

2.6.7 rewrap

```
matrix = m:rewrap(size1, size2, ...)
```

This method only works if the data is contiguous in memory. The caller matrix is reinterpreted as if it was of another number of dimensions and sizes. A different matrix instance is returned, but the data pointer is shared.

```
> a = matrix(2,3,{1,2,3, 4,5,6})
> = a
1 2 3
4 5 6
```

```
# Matrix of size [2,3] in row_major [0x2700850 data= 0x2700900]
> b = a:rewrap(3,2)
> = b
1 2
3 4
5 6
# Matrix of size [3,2] in row_major [0x2701360 data= 0x2700900]
```

2.6.8 select

`matrix = m:select(dimension, index [, matrix])`

This methods returns a matrix with one less dimension, resulting of select at the caller matrix the indicated dimension at the given index. The resulting matrix references the internal data of original matrix. If given, the **third argument** must be a `matrix` which will be used to store the result of the `select` call, and must fit the expected dimensions. In this last case, the computation effort is dismissed to constant.

```
> m = matrix(4,3):zeros()
> = m
0 0 0
0 0 0
0 0 0
0 0 0
# Matrix of size [4,3] [0x23dcab0 data= 0x23727e0]
> = m:select(2,2):fill(9)
9 9 9 9
# Matrix of size [4] [0x23dd330 data= 0x23727e0]
> = m:select(1,3):fill(4)
4 4 4
# Matrix of size [3] [0x23dd790 data= 0x23727e0]
> = m
0 9 0
0 9 0
4 4 4
0 9 0
# Matrix of size [4,3] [0x23dcab0 data= 0x23727e0]
```

NOTE that the third argument matrix must be created by a previous call to `select` over the **same** dimension (but not the same index). As example, the following design pattern gives the same variable as third argument result and as left-side of the expression, allocating the memory in the first loop iteration, and reusing it in the following:

```
> m = matrix(4,5):linear()
> for i=1,m:dim(2) do
    result = m:select(2,i,result)
  end
```

2.6.9 transpose

`matrix = m:transpose()`

This method returns a `matrix` which is a transposition of the caller object. Note that both, the caller and the transposition, reference the same data.

```
> m = matrix(3,4):linear()
> = m
 0           1           2           3
 4           5           6           7
 8           9           10          11
# Matrix of size [3,4] in row_major [0x2777140 data= 0x27799b0]
> = m:transpose()
 0           4           8
 1           5           9
 2           6           10
 3           7           11
# Matrix of size [4,3] in row_major [0x274e620 data= 0x27799b0]
```

2.6.10 slice

`matrix = m:slice(position, size [, clone])`

This methods produces a sub-matrix of the caller matrix. By default, the returned sub-matrix shares the data pointer with the caller, but it is also possible to do a deep copy sub-matrix. The syntax is:

```
m:slice(pos_table, size_table, clone=false)
```

being `pos_table` a Lua table with the position of first element (starting at 1, not 0), and `size_table` a Lua table with the size of each dimension. The last argument, `clone`, is an optional boolean (by default `false`) indicating if the resulting matrix will be a clone or not.

```
> a = matrix(3,4,{1,2,3,4, 5,6,7,8, 10,11,12,13}) -- row major matrix
> = a
1 2 3 4
5 6 7 8
10 11 12 13
# Matrix of size [3,4] in row_major [0x2706530 data= 0x2706b00]
> b = a:slice({2,1},{2,2}) -- slice at position (2,1) with size 2x2
> = b
5 6
10 11
# Matrix of size [2,2] in row_major [0x2707cd0 data= 0x2706b00]
> -- same slice as before but making a clone (deep copy)
> b = a:slice({2,1},{2,2}, true)
> = b
5 6
10 11
# Matrix of size [2,2] in row_major [0x2708a20 data= 0x2708ad0]
```

2.6.11 operator[]

2.6.11.1 Right hand side operator (___index)

`matrix = m[number]`

A shortcut for *select* or *get* methods. The operator is equivalent to a `m:select(1,n)` in case the matrix has more than one dimension, and returns a `matrix` object with one less dimension. In case of uni-dimensional matrix, this call is equivalent to `m:get(n)` and returns a Lua number value.

This operator can be used in left and/or right hand of an assignment. In the right hand:

- `m[key] = number` is equivalent to `m[key]:fill( number )` in case the right-hand is a number.
- `m[key] = matrix` is equivalent to `m[key]:copy( matrix )` in case the right-hand is a `matrix`.

`matrix = m[{s1, s2, ...}]`

A shortcut for *slicing*, using the `__index` metamethod. It is similar to Matlab or Octave slice operators. The call is parsed and converted into a `slice` method call, so, it is **slower** than using directly the `slice` method, but it is easier to understand. This operator receives a variable number of arguments, as many as dimensions has the caller `matrix` object. In every dimension position, three possible values could be given:

- A number, indicating an exact dimension position: `m = m2[{2,3}]`
- A table, with start and end positions in the dimension: `m = m2[{2, {1,3}}]`
- A string, with start and end positions in the dimension: `m = m2[{2, '1:3'}]`

The following examples are equivalent:

```
> = m2:slice({2,1},{1,3})
 4           5           6
# Matrix of size [1,3] in row_major [0xf44470 data= 0xdeae90]
> = m2[{2, {1,3}}]
 4           5           6
# Matrix of size [1,3] in row_major [0xf3c8d0 data= 0xdeae90]
> = m2[{2, '1:3'}]
 4           5           6
# Matrix of size [1,3] in row_major [0xe32dc0 data= 0xdeae90]
```

It is possible to use the string shortcut to indicate a whole dimension, or only start/end positions:

```
> = m2[{':', '2:3'}]
 2           3
 5           6
 8           9
# Matrix of size [3,2] in row_major [0xef1260 data= 0xdeae90]
> = m2[{'2:', '2:3'}]
 5           6
 8           9
# Matrix of size [2,2] in row_major [0xe08f50 data= 0xdeae90]
> = m2[{':2', '2:3'}]
 2           3
 5           6
# Matrix of size [2,2] in row_major [0xf04290 data= 0xdeae90]
```

2.6.11.2 Left hand side operator (___newindex)

`m[key] = value`

Different cases depending in the type of `key` and the type of `value`.

- key=number, value=number: it is equivalent to `m:select(1,key):fill(value)`

- key=number, value=matrix: it is equivalent to `m:select(1,key):copy(value)`
- key=table, value=number: it is equivalent to `m[key]:fill(value)`
- key=table, value=matrix: it is equivalent to `m[key]:copy(value)`
- key=matrixBool, value=number: it is equivalent to `m:masked_fill(key, value)`
- key=matrixBool, value=matrix: it is equivalent to `m:masked_copy(key, value)`

```
> -- assigns a 0 to all the row values at columns 2:3
> m2[{ ':', '2:3' }] = 0
> -- assigns another matrix all the row values at columns 2:3
> m2[{ ':', '2:3' }] = matrix(m2:dim(1),2):linspace()
```

2.6.12 join

`matrix = matrix.join(dimension, m1, m2, ...)`

This function joins the given matrices by the given dimension. All the dimensions of the matrices must be the same, except the given dimension, which could differ. It is possible to add a new axis at left and join all matrices using the new added axis given a `dimension=0` parameter. Similarly, it is possible to add a new axis at right for joining all matrices using this new axis given a `dimension=num_dims+1`, being num_dim the number of dimensions of the given matrices.

Warning, this method duplicates the memory needed, because all the matrices are copied to the destination matrix.

```
> m1 = matrix(10,2):linear()
> m2 = matrix(10,3):linear()
> outm = matrix.join(2, m1, m2)
> = outm
 0           1           0           1           2
 2           3           3           4           5
 4           5           6           7           8
 6           7           9           10          11
 8           9           12          13          14
 10          11          15          16          17
 12          13          18          19          20
 14          15          21          22          23
 16          17          24          25          26
 18          19          27          28          29
# Matrix of size [10,5] in row_major [0x1f9c100 data= 0x1f9c1c0]
>
> m1 = matrix(10,3):linear()
> -- m2 remains the same
> -- add new axis at left
> = matrix.join(0, m1, m2)
# pos [1,1,1]
 0           1           2
 3           4           5
 6           7           8
 9           10          11
 12          13          14
...
 27          28          29
```

```
# pos [2,1,1]
 0              1              2
 3              4              5
 6              7              8
 9              10             11
 12             13             14
...
 27             28             29
# Matrix of size [2,10,3] stride [30,3,1] ref [0x1940f90 data= 0x177c280]
>
> -- add new axis at right
> = matrix.join(3, m1, m2)

# pos [1,1,1]
 0              0
 1              1
 2              2

# pos [2,1,1]
 3              3
 4              4
 5              5

...

# pos [10,1,1]
 27             27
 28             28
 29             29
# Matrix of size [10,3,2] stride [6,2,1] ref [0x1b42ef0 data= 0x1b42fe0]
```

2.6.13 clamp

```
matrix = m:clamp(lower, upper)
```

This method clamps the matrix components to a given range [min,max], modifying the matrix **in-place**. The caller matrix instance is returned.

```
> a = matrix(3,3,{1,2,3,4,5,6,7,8,9})
> = a
1 2 3
4 5 6
7 8 9
# Matrix of size [3,3] in row_major [0xe56a30 data= 0xe56f40]
> a:clamp(3,6)
> = a
3 3 3
4 5 6
6 6 6
# Matrix of size [3,3] in row_major [0xe56a30 data= 0xe56f40]
```

2.6.14 adjust_range

```
matrix = m:adjust_range(min, max)
```

This method modifies **in-place** the matrix components, interpolating the values to be in the given range [min,max]. The caller matrix is returned.

```
> a = matrix(3,3,{1,2,3,4,5,6,7,8,9})
> a:adjust_range(3,6)
> = a
3 3.375 3.75
4.125 4.5 4.875
5.25 5.625 6
# Matrix of size [3,3] in row_major [0x25cca30 data= 0x25ccf40]
> = a:adjust_range(0,1)
0 0.125 0.25
0.375 0.5 0.625
0.75 0.875 1
# Matrix of size [3,3] in row_major [0x25cca30 data= 0x25ccf40]
> = a:adjust_range(1,9)
1 2 3
4 5 6
7 8 9
# Matrix of size [3,3] in row_major [0x25cca30 data= 0x25ccf40]
```

2.6.15 is_contiguous

```
boolean = m:is_contiguous()
```

Indicates if the matrix internal data is contiguous at memory (in row major order).

2.6.16 contiguous

```
matrix = m:contiguous()
```

Returns a contiguous version of the caller matrix. If the matrix is contiguous, returns itself. Otherwise, returns a copy of the caller. **Note** that, if the matrix is a slice or a transposition of another matrix, therefore it could be non-contiguous.

2.7 Data initialization methods

2.7.1 fill

```
matrix = m:fill(number)
```

This is an **in-place** method which sets all components to a given value.

```
> a = matrix(2,3):fill(4) -- a 2x3 matrix filled with 4
> = a
4 4 4
4 4 4
# Matrix of size [2,3] in row_major [0x26ff9b0 data= 0x26ffa20]
```

2.7.2 zeros

```
matrix = m:zeros()
```

This is equivalent to `m:fill(0)`

2.7.3 ones

```
matrix = m:ones()
```

This is equivalent to `m:fill(1)`

2.7.4 linear

```
matrix = m:linear(start=0, step=1)
```

Initializes the matrix starting at the given index and using the given step. The index and the step is optional.

```
> m = matrix(3,2,2):linear(1,2)
> = m

# pos [1,1,1]
1 3
5 7

# pos [2,1,1]
9 11
13 15

# pos [3,1,1]
17 19
21 23
# Matrix of size [3,2,2] in row_major [0x149de00 data= 0x149daa0]
> m = matrix(2,2):linear()
> = m
0 1
2 3
# Matrix of size [2,2] in row_major [0x149f110 data= 0x149f1e0]
```

2.7.5 linspace

```
matrix = m:linspace(a=1, b=m:size())
```

Initializes the `matrix` with a linear space distribution. It receives two **optional** arguments, why default `a=1` and `b=m:size()`. It returns the caller `matrix`.

```
> m = matrix(5):linspace(1,20)
> = m
 1          5.75        10.5        15.25       20
# Matrix of size [5] in row_major [0x291f200 data= 0x291ecd0]
```

2.7.6 logspace

```
matrix = m:logspace(a=1, b=m:size(), base=10)
```

Initializes the `matrix` with a logarithmic distribution between `a` and `b` with the given logarithm `base`. It receives three **optional** arguments, why default `a=1`, `b=m:size()` and `base=10`. It returns the caller `matrix`.

```
> m = matrix(5):logspace(0.001,0.1)
> = m
 0.001       0.00316228  0.01        0.0316228   0.1
# Matrix of size [5] in row_major [0x291fed0 data= 0x291fd50]
```

2.7.7 uniform

```
matrix = m:uniform(lower, upper [, random] )
```

This method initializes the matrix with random *integers* taken uniformly from the given range of values:

```
> m = matrix(10):uniform(0,10,random(1234))
> = m
3 6 5 4 8 9 1 7 9 10
# Matrix of size [10] in row_major [0x2716b10 data= 0x2716490]
```

The random object is optional, but to ensure reproducibility it is recommended.

2.7.8 uniformf

```
matrix = m:uniformf(lower=0, upper=1 [, random] )
```

This method initializes the matrix with random *floats* taken uniformly from the given range of values:

```
> m = matrix(2,2):uniformf(-10, 10, random(1234))
> = m
-6.16961 -0.0467267
2.44218 6.35677
# Matrix of size [2,2] in row_major [0x1000e90 data= 0xe47410]
```

The random object is optional, but to ensure reproducibility it is recommended.

2.7.9 diag

```
matrix = m:diag(number)
```

This method sets the matrix diagonal components to a given value, modifying **in-place** the caller matrix. For any number of dimensions, the diagonal are whose components which positions are equals at all dimensions.

```
> a = matrix(3,3,3):ones():diag(5)
> = a

# pos [1,1,1]
5 1 1
1 1 1
```

```
1 1 1

# pos [2,1,1]
1 1 1
1 5 1
1 1 1

# pos [3,1,1]
1 1 1
1 1 1
1 1 5
# Matrix of size [3,3,3] in row_major [0x1718f10 data= 0x1718d50]
```

2.8 Matrix serialization

2.8.1 toString

`string = m:toString( mode='ascii' )`

This method returns a Lua string which represents the caller matrix. It receives an optional argument indicating if the matrix data will be stored in `ascii` or `binary` format (by default `ascii`).

```
> a = matrix(3,5):ones()
> = a
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
# Matrix of size [3,5] in row_major [0xd80a10 data= 0xd815d0]
> = a:toString()
3 5
ascii
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

> = a:toString("ascii")
3 5
ascii
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

> = a:toString("binary")
3 5
binary
8Ffe98Ffe98Ffe98Ffe98Ffe98Ffe98Ffe98Ffe98Ffe98Ffe98Ffe98Ffe98Ffe98Ffe98Ffe98Ffe9
```

2.8.2 fromString

`matrix = matrix.fromString(filename)`

This method loads a matrix from a Lua string generated by method `matrix.toString`.

```
> a = matrix.fromString[[3 5
>> ascii
```

```
>> 1 1 1 1 1 1 1 1 1
>> 1 1 1 1 1 1
>> ]]
> = a
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
# Matrix of size [3,5] in row_major [0xd983b0 data= 0xdfe5c0]
```

2.8.3 toFilename

`m:toFilename(filename, mode='ascii')`

This method stores a matrix in a given filename. It also receives an optional argument with `ascii` or `binary` (by default `ascii`). It allows to compress the output file using GZIP, if the filename has '.gz' extension.

```
> a = matrix(3,3)
> a:toFilename("a.mat", "binary")
> a:toFilename("a.mat.gz", "binary")
```

2.8.4 fromFilename

`matrix = matrix.fromFilename(filename)`

This method loads a matrix from a given filename, expecting the format used by `matrix.toFilename` method. It allows to load compressed files using GZIP, if the filename has '.gz' extension.

```
> a = matrix.fromFilename("a.mat")
> a = matrix.fromFilename("a.mat.gz")
```

2.8.5 toTabFilename

`m:toTabFilename(filename)`

This method stores a matrix in a given filename, but without header, and formatting the data to be formatted by lines and spaces, one matrix row per line. It is limited to **bi-dimensional matrices**. It allows to compress the output file using GZIP, if the filename has '.gz' extension.

```
> a = matrix(3,3)
> a:toTabFilename("a.mat")
> a:toTabFilename("a.mat.gz")
```

2.8.6 fromTabFilename

`matrix = matrix.fromTabFilename(filename)`

This method loads a matrix from a given filename, formatted as done by `matrix.toTabFilename`. The size of the `matrix` is computed in a first loop over all the data, so this method needs two passes to load the matrix. It allows to load compressed files using GZIP, if the filename has '.gz' extension.

```
> a = matrix.fromTabFilename("a.mat")
> a = matrix.fromTabFilename("a.mat.gz")
```

2.8.7 toMMap

`m:toMMap(filename)`

Stores the `matrix` in a file in a binary machine-dependent format, so it could be loaded using mmap function (`matrix.fromMMap`). The endianism must be the same between machines where matrix is stored/loaded.

2.8.8 matrix.fromMMap

`matrix = matrix.fromMMap(filename [,write [,shared]])`

Loads the `matrix` from a file in a binary machine-dependent format, by using the mmap function (`matrix.toMMap`). The endianism must be the same between machines where matrix is stored/loaded. *Two additional boolean* arguments are allowed. The **second** boolean argument indicates if *writing* is available, by default it is `true`. Be careful, if writing is set to `false`, any attempt of writing will throw a segmentation fault. The **third** boolean argument indicates if the data is *shared* between different processes, by default it is `true`. If both arguments are `true`, any writing will be available to any process which shares this map. Besides, writings will be synchronized in the hard disk (but not instantly). If writing is `true`, but shared is `false`, then the memory is mapped as *copy-on-write*. For more info, see the manual page of mmap function (PROTECT_WRITE, MAP_SHARED and MAP_PRIVATE).

2.8.9 toTable

`table = m:toTable()`

This method returns a plain Lua table (one-dimensional table) which contains the matrix data in `row_major` order, as expected by `matrix` constructors.

```
> a = matrix(3,2,{1,2,3,4,5,6})
> = a
1 2
3 4
5 6
# Matrix of size [3,2] in row_major [0x9ddce0 data= 0x9ddd30]
> t = a:toTable()
> = table.concat(t, " ")
1 2 3 4 5 6
```

2.8.10 read

2.8.11 write

2.9 Low-level matrix access

These methods allows raw accessing of matrix components.

2.9.1 size

`number = m:size()`

This method returns the number of elements in the matrix.

```
> a = matrix(3,4,{1,2,3,4, 5,6,7,8, 10,11,12,13})
> = a:size()
12
```

2.9.2 stride

`table = m:stride()`

This method is similar to `m:dim`, but returning the stride of the dimension (the offset between elements at each dimension)

```
> a = matrix(4,3,2)
> = a:stride()
table: 0x23a5fe0
> = table.concat(a:stride(), " ")
6 2 1
> = a:stride(1), a:stride(2), a:stride(3)
6   2   1
> a = a:transpose()
> = a:stride(1), a:stride(2), a:stride(3)
1   4   12
```

2.9.3 offset

`number = m:offset()`

It returns the offset from data first position. Only sub-matrices has an offset!=0.

```
> a = matrix(2,3)
> = a:offset()
0
> b = a:slice({2,1},{1,1})
> = b:offset()
3
```

2.9.4 data

`block = m:data()`

Returns a float memory block (`mathcore.block.float`) with the underlying data pointer.

2.9.5 raw_get

`number = m:raw_get(pos)`

It receives a raw position at the underlying data pointer, and returns its value. It is useful to combine `stride` and `offset` methods in order to compute the raw position.

```
> a = matrix(3,2, {1,2,3,4,5,6})
> = a
1 2
3 4
```

```
5 6
# Matrix of size [3,2] in row_major [0x144fce0 data= 0x144fd90]
> = a:raw_get(a:offset() + a:stride(1)*1 + a:stride(2)*0), a:get(2,1)
3   3
```

NOTE! that the strides are multiplied by matrix position minus 1.

2.9.6 raw_set

`m:raw_set(pos, value)`

It receives a raw position at the underlying data pointer and a number. The given position is set to given number value. It is useful to combine `stride` and `offset` methods in order to compute the raw position.

```
> a = matrix(3,2, {1,2,3,4,5,6})
> = a
1 2
3 4
5 6
# Matrix of size [3,2] in row_major [0x144fce0 data= 0x144fd90]
> -- equivalent to a:set(2,1, 10)
> a:raw_set(a:offset() + a:stride(1)*1 + a:stride(2)*0, 10)
> = a
1 2
10 4
5 6
# Matrix of size [3,2] in row_major [0x144fce0 data= 0x144fd90]
```

NOTE! that the strides are multiplied by matrix position minus 1.

2.10 Sliding window iterator

For fast and easy matrix traversal, a C++ sliding window object is binded to Lua. It works similarly to `dataset.matrix`, but circularity and out-of-matrix default values are not supported. The object is constructed using the method `sliding_window` of `matrix`, and could be iterated using its method `iterate()`:

```
> m = matrix(4,2,3):uniformf(-10,10,random(1234)) -- randomly initialized matrix
> for submat in m:sliding_window():iterate() do print(submat) end

# pos [1,1,1]
-6.16961 -0.0467267 2.44218
6.35677 -1.24545 2.24224
# Matrix of size [1,2,3] in row_major [0x253f160 data= 0x253dec0]

# pos [1,1,1]
5.70717 5.4272 5.59952
7.2134 -4.54815 -6.98726
# Matrix of size [1,2,3] in row_major [0x253fa40 data= 0x253dec0]

# pos [1,1,1]
-4.47071 -6.02962 6.03744
```

```
6.30326 9.16279 -6.82369
# Matrix of size [1,2,3] in row_major [0x2540230 data= 0x253dec0]

# pos [1,1,1]
7.51865 -7.67724 -2.84365
-9.74185 0.0199025 -0.263331
# Matrix of size [1,2,3] in row_major [0x25409c0 data= 0x253dec0]
```

It is possible to modify the default behavior giving this parameters to `sliding_window` method:

- `offset`: a Lua table with offset applied to the window in each coordinate (starting at 0).
- `size`: a Lua table with the window size for each coordinate.
- `step`: a Lua table with the step size at each coordinate (each value must be >= 1).
- `numSteps`: a Lua table with the number of steps in each coordinate (each value must be >= 1).
- `orderStep`: a Lua table with the traversal order of coordinates (starting at 1).

```
> m = matrix(4,2,3):uniformf(-10,10,random(1234))
> for w in m:sliding_window{ step={2,1,1}, size={1,1,2} }:iterate() do print(w) end
# pos [1,1,1]
-6.16961 -0.0467267
# Matrix of size [1,1,2] in row_major [0x9fdb90 data= 0x9cf2d0]

# pos [1,1,1]
-4.47071 -6.02962
# Matrix of size [1,1,2] in row_major [0x9fe0f0 data= 0x9cf2d0]
```

Manual iteration of the sliding_window is also possible using the following methods:

- `matrix = sw:get_matrix([matrix])`: returns the matrix generated by the window at its current position. It is possible to pass an **optional argument**, a destination `matrix`, so the computation effort is dismissed to constant. **NOTE** that this matrix must be created by a previous call to `get_matrix` over the **same** `sliding_window`.
- `sw:next()`: moves the window to the next position.
- `sw:is_end()`: returns true if the window has finished the matrix traversal.

```
> m = matrix(4,2,3):uniformf(-10,10,random(1234))
> wo = m:sliding_window{ step={2,1,1}, size={1,1,2} }
> while not wo:is_end() do m=wo:get_matrix(m) print(m) wo:next() end
```

2.11 Fast mathematical operations

This operations uses standard Lua math operators for friendly user interaction, but they work with BLAS API for best performance. However, all this operations return a new instantiated matrix, for best performance it is recommended to use directly the BLAS interface.

The operators binary +, -, *, /, and unary operators -, ^, are implemented as algebraic operations. The + and - operators only work when the matrices has the same sizes:

```
> a= matrix(3,3,3,{1,2,3,4,5,6,7,8,9,
                   10,11,12,13,14,15,16,17,18,
                   19,20,21,22,23,24,25,26,27})
> = a+a

# pos [1,1,1]
2 4 6
8 10 12
14 16 18

# pos [2,1,1]
20 22 24
26 28 30
32 34 36

# pos [3,1,1]
38 40 42
44 46 48
50 52 54
# Matrix of size [3,3,3] in row_major [0x1196d90 data= 0x1196e40]
> = a-a

# pos [1,1,1]
0 0 0
0 0 0
0 0 0

# pos [2,1,1]
0 0 0
0 0 0
0 0 0

# pos [3,1,1]
0 0 0
0 0 0
0 0 0
# Matrix of size [3,3,3] in row_major [0x1198d80 data= 0x1198a50]
```

The operator * only works with vectors or bi-dimensional matrices. If needed, you can `rewrap` the matrix data before the operation. Depending on the dimension of the two matrices, the multiplication could be:

- A **dot product** between two vectors: when the two matrices are unidimensional vectors, the first one a vector and the second one a column vector:

```
> a, b = matrix(4,{1,2,3,4}), matrix(4,1,{5,6,7,8})
> = a*b
70
# Matrix of size [1] in row_major [0xfa9230 data= 0xfc2300]
```

- An **outter product** between two vectors: when the first matrix is a column vector, and the second matrix is a unidimensional matrix or a bi-dimensional matrix (row or column vector).

```
> a = matrix(4,{1,2,3,4})
> b = matrix(4,1,{5,6,7,8})
```

```
> = b*a
5 10 15 20
6 12 18 24
7 14 21 28
8 16 24 32
# Matrix of size [4,4] in row_major [0x1001940 data= 0x1176a80]
```

- A **matrix-vector product** when the first matrix is a bi-dimensional matrix and the second is a vector. The output has the same number of dimensions as the given vector.

```
> a = matrix(2,2,{1,2,3,4})
> b = matrix(2,{5,6})
> = a*b
17 39
# Matrix of size [2] in row_major [0x105baa0 data= 0xfe80f0]
> b = matrix(1,2,{5,6})
> = a*b
17
39
# Matrix of size [2,1] in row_major [0x107e3c0 data= 0x107fb30]
> b = matrix(2,1,{5,6})
> = a*b
17
39
# Matrix of size [2,1] in row_major [0x10c4700 data= 0x10c6890]
```

- A **matrix-matrix product** when the two matrices are bi-dimensional and not vectors.

```
> a=matrix(3,2,{1,2,3,4,5,6})
> b=matrix(2,4,{1,2,3,4,5,6,7,8})
> = a*b
11 14 17 20
23 30 37 44
35 46 57 68
# Matrix of size [3,4] in row_major [0x1114270 data= 0x11165d0]
```

A multiplication by a scalar is also possible, if you multiply one matrix by one number.

```
> a=matrix(3,2,{1,2,3,4,5,6})
> = a*5
5 10
15 20
25 30
# Matrix of size [3,2] in row_major [0x10f2160 data= 0x10d14e0]
```

The component-wise operator / is allowed for division between **matrix** and a **scalar**, or between a **scalar** and a **matrix**.

The operator ^ is also allowed only with **scalars**.

The unary operator - is equivalent to multiply by -1.

2.11.1 scalar_add

`matrix = m:scalar_add(number)`

Adds to all the components, **in-place**, the given scalar number. Returns the caller matrix object.

2.11.2 div

`matrix = m:div(scalar)`

Produces the computation between the component-wise inversion of the `matrix` and the given scalar. This operation is done **in-place**.

```
> m = matrix(2,2,{1,2,3,4})
> m:div(1)
> = m
1      0.5
0.3333 0.25
# Matrix of size [3,2] in row_major [0x1cf2160 data= 0x10d15e0]
```

2.12 BLAS interface

The most efficient way to do operations if using the BLAS interface directly. All the methods are prepared to adjust the BLAS operations to the given matrices, so you don't need to be worried about strides and sizes.

All of this methods are **in-place**, so they modify the caller object, and returns it to allow operation sequences.

2.12.1 axpy

`matrix = m:axpy(alpha, X)`

The AXPY operation computes addition of vectors:

`Y = alpha * X + Y`

The method receives two positional parameters: the alpha scalar and the matrix X. The X and Y matrix sizes must be equals, and the number of dimensions is not a problem. This method interprets all the data as a sequence, calling several times to AXPY BLAS function if necessary:

```
> a = matrix(4,{1,2,3,4})
> b = matrix(4,{5,6,7,8})
> a:axpy(2.0, b)
> = a
11 14 17 20
# Matrix of size [4] in row_major [0x107e3c0 data= 0x1110970]
> a = matrix(2,2,2,{1,2,3,4,5,6,7,8})
> b = matrix(2,2,2,{9,10,11,12,13,14,15,16})
> a:axpy(1.0, b)
> = a

# pos [1,1,1]
10 12
14 16
```

```
# pos [2,1,1]
18 20
22 24
# Matrix of size [2,2,2] in row_major [0xfb1f40 data= 0x1056f00]
```

2.12.2 gemv

`matrix = m:gemv{ beta, alpha, Y, X, trans_A}`

The GEMV operation computes matrix-vector multiplication:

`Y = beta * Y + alpha * op(A) * X`

being Y the caller matrix (a vector), A another matrix, and X a vector (unidimensional matrix, or bi-dimensional with one row (or one column)), and beta and alpha are scalars. The op(A) is transposition operation.

The method receives a table with:

- `A` field, the other matrix.
- `X` field, the vector.
- `alpha` field, the scalar
- `beta` field, the other scalar.
- `trans_A` field, a boolean which indicates if the A matrix will be transposed or not. It is optional, by default is `false`.

```
> a = matrix(3,2,{1,2, 3,4, 5,6})
> b = matrix(2,{7,8})
> c = matrix(3)
> c:gemv{ A=a, X=b, alpha=2, beta=0 }
> = c
46 106 166
# Matrix of size [3] in row_major [0xfbeff0 data= 0xfaf640]
```

2.12.3 gemm

`matrix = m:gemm{ beta, alpha, A, B, ... }`

The GEMM operation computes matrix-matrix multiplication:

`Y = beta * Y + alpha * op(A) * op(B)`

being Y the caller matrix (a vector), A another matrix, and B a matrix, and beta and alpha are scalars. The op(A) and op(B) are transposition operations.

The method receives a table with:

- `A` field, the other matrix.
- `B` field, the vector.
- `alpha` field, the scalar

- `beta` field, the other scalar.
- `trans_A` field, a boolean which indicates if the A matrix will be transposed or not. It is optional, by default is `false`.
- `trans_B` field, a boolean which indicates if the B matrix will be transposed or not. It is optional, by default is `false`.

```
> a = matrix(3,2,{1,2, 3,4, 5,6})
> b = matrix(4,2,{7,8, 9,10, 11,12, 13,14})
> c = matrix(3,4):ones()
> c:gemm{ A=a, B=b, alpha=1, beta=1, trans_B=true}
> = c
24 30 36 42
54 68 82 96
84 106 128 150
# Matrix of size [3,4] in row_major [0x1452a20 data= 0x144cbf0]
```

2.12.4 ger

`matrix = m:ger{ X, Y, alpha }`

The GER operation computes outter product of vectors:

`Z = Z + alpha * X * Y'`

being Z the caller matrix (a squared matrix), X and Y two vectors, and beta and alpha are scalars. The Y vector is transposed.

```
> a = matrix(3,{1,2,3})
> b = matrix(3,{4,5,6})
> c = matrix(3,3):zeros()
> c:ger{ X=a, Y=b, alpha=2 }
> = c
8 10 12
16 20 24
24 30 36
# Matrix of size [3,3] in row_major [0x1f06b20 data= 0x1f18080]
```

2.12.5 dot

`number = m:dot(matrix)`

The DOT operation computes the dot-product of two vectors, the caller matrix and a given matrix. It returns a number.

```
> a = matrix(3,{1,2,3})
> b = matrix(3,{4,5,6})
> = a:dot(b)
32
# Matrix of size [1] in row_major [0x1f4ffe0 data= 0x2076e20]
```

2.12.6 scal

```
matrix = m:scal(number)
```

The SCAL operation computes the multiplication of a matrix by a scalar.

```
> a = matrix(3,{1,2,3})
> a:scal(4)
> = a
4 8 12
# Matrix of size [3] in row_major [0x1f3b230 data= 0x201e9a0]
```

2.12.7 copy

```
matrix = m:copy(matrix)
```

The COPY operation copies the content of a given matrix in the caller matrix object.

```
> a = matrix(3,3,{1,2,3,4,5,6,7,8,9})
> b = matrix(3,3):fill(5)
> a:copy(b)
> = a
5 5 5
5 5 5
5 5 5
# Matrix of size [3,3] in row_major [0x1f7e870 data= 0x1f49ef0]
> a = matrix(3,3,{1,2,3,4,5,6,7,8,9})
> b = matrix(2,2,{1,2,3,4})
> c = a:slice({2,1},{2,2})
> c:copy(b)
> = a
1 2 3
1 2 6
3 4 9
# Matrix of size [3,3] in row_major [0x1fb64e0 data= 0x1fbd600]
```

2.13 LAPACK interface

2.13.1 svd

```
U,S,VT = m:svd()
```

This method computes the Singular Values Decomposition of the caller `matrix`. It returns three matrices:

- `U` a matrix with the left singular vectors.
- `S` a sparse diagonal matrix with singular values.
- `VT` a matrix with the transposed right singular vectors.

2.13.2 inv

```
matrix = m:inv()
```

Computes the inverse of the caller matrix. Check that your matrix is not singular, otherwise the returned matrix won't be correct.

2.13.3 pinv

```
matrix = m:pinv()
```

Computes the pseudo-inverse of the caller matrix, using the SVD method.

2.14 Component-wise operations

This operations are applied **in-place** and over all the components of the caller matrix. If it is possible, the caller matrix is returned.

2.14.1 tan

```
matrix = m:tan()
```

Computes the TAN function of all the components.

2.14.2 tanh

```
matrix = m:tanh()
```

Computes **in-place** the TANH function of all the components.

2.14.3 atan

```
matrix = m:atan()
```

Computes **in-place** the ATAN function of all the components.

2.14.4 atanh

```
matrix = m:atanh()
```

Computes **in-place** the ATANH function of all the components.

2.14.5 sin

```
matrix = m:sin()
```

Computes **in-place** the SIN function of all the components.

2.14.6 sinh

```
matrix = m:sinh()
```

Computes **in-place** the SINH function of all the components.

2.14.7 asin

```
matrix = m:asin()
```

Computes **in-place** the ASIN function of all the components.

2.14.8 asinh

```
matrix = m:asinh()
```

Computes **in-place** the ASINH function of all the components.

2.14.9 cos

```
matrix = m:cos()
```

Computes **in-place** the COS function of all the components.

2.14.10 cosh

```
matrix = m:cosh()
```

Computes **in-place** the COSH function of all the components.

2.14.11 acos

```
matrix = m:acos()
```

Computes **in-place** the ACOS function of all the components.

2.14.12 acosh

```
matrix = m:acosh()
```

Computes **in-place** the ACOSH function of all the components.

2.14.13 abs

```
matrix = m:abs()
```

Computes **in-place** the ABS function of all the components.

2.14.14 complement

```
matrix = m:complement()
```

Computes **in-place** the complement function of all the components: X = 1 - X

2.14.15 log

```
matrix = m:log()
```

Computes **in-place** the LOG function of all the components.

2.14.16 log1p

```
matrix = m:log1p()
```

Computes **in-place** the LOG1P function of all the components.

2.14.17 plogp

```
matrix = m:plogp()
```

Computes **in-place** the p*log(p) operation over all components. It is useful to compute entropy related measures.

2.14.18 exp

```
matrix = m:exp()
```

Computes **in-place** the EXP function of all the components.

2.14.19 pow

```
matrix = m:pow(number)
```

Computes **in-place** the POWER of all the components by a given scalar.

2.14.20 sqrt

```
matrix = m:sqrt()
```

Computes the SQRT function of all the components.

2.14.21 cmul

```
matrix = m:cmul(matrix)
```

Computes **in-place** a component-wise multiplication between the caller and a given matrix.

2.15 Matrix level operations

This operations are applied taking into account all the data at the matrix.

2.15.1 min

```
min,argmin = m:min()
```

Returns the minimum and its position in the matrix.

```
> a = matrix(3,4,{1,2,3,4,5,6,7,12,9,10,11,8})
> = a:min()
1   1
```

```
matrix,matrixInt32 = m:min(dim [, matrix[, matrixInt32]] )
```

Applies the min operator over the elements of the given dimension, and returns a matrix with the same number of dimensions, but with the size of dimension `dim` equals 1. The second matrix argument is optional, and if given, the returned matrix will be this second argument.

```
> a = matrix(3,4,{1,2,3,4,
>>                5,6,7,12,
>>                9,10,11,8})
> = a:min(1)
 1           2           3           4
# Matrix of size [1,4] in row_major [0x1f06bb0 data= 0x1f06cb0]
> = a:min(2)
 1
 5
 8
# Matrix of size [3,1] in row_major [0x1f07560 data= 0x1f06d90]
```

2.15.2 max

`max,argmax = m:max()`

Returns the maximum and its position in the matrix.

```
> a = matrix(3,4,{1,2,3,4,5,6,7,12,9,10,11,8})
> = a:max()
12  8
```

`matrix,matrixInt32 = m:max(dim [, matrix[, matrixInt32] ] )`

Applies the max operator over the elements of the given dimension, and returns a matrix with the same number of dimensions, but with the size of dimension `dim` equals 1. The second matrix argument is optional, and if given, the returned matrix will be this second argument.

```
> a = matrix(3,4,{1,2,3,4,
>>                5,6,7,12,
>>                9,10,11,8})
> = a:max(1)
 9           10          11          12
# Matrix of size [1,4] in row_major [0x1f05500 data= 0x1f05600]
> = a:max(2)
 4
 12
 11
```

2.15.3 eq

`matrixBool = m:eq(number or matrix)`

This method computes **in-place** a comparison between all the components with the given value, and converts the component in `true` or `false` if it is less than the given value or not. If the value is another `matrix`, both matrices will be compared component-by-component.

2.15.4 lt

`matrixBool = m:lt(number or matrix)`

This method computes **in-place** a comparison between all the components with the given value, and converts the component in `true` or `false` if it is less than the given value or not. If the value is another `matrix`, both matrices will be compared component-by-component.

2.15.5 gt

```
matrixBool = m:gt(number or matrix)
```

This method computes **in-place** a comparison between all the components with the given value, and converts the component in `true` or `false` if it is greater than the given value or not. If the value is another `matrix`, both matrices will be compared component-by-component.

2.15.6 sum

```
number = m:sum( )
```

Computes the sum of all the components of the caller matrix, and returns its value.

```
matrix = m:sum( number [, matrix] )
```

Receives a number indicating the dimension where the sum must be run, and returns a matrix with each possible sum of the given dimension. The second matrix argument is optional, and if given, the returned matrix will be this argument.

```
> m = matrix(2,2,{1,2,3,4})
> = m:sum(1)
 4           6
# Matrix of size [1,2] in row_major [0x19e0620 data= 0x19d2480]
> = m:sum(2)
 3
 7
# Matrix of size [2,1] in row_major [0x19e0a40 data= 0x19d3b90]
```

2.15.7 norm2

```
number = m:norm2()
```

The NORM2 operation computes the euclidean norm of the caller matrix. It returns a number.

```
> a = matrix(2,2,2,{1,2,3,4,5,6,7,8})
> = a:norm2()
14.282856941223
```

2.16 Indexing and sorting

This operations allow to extract or sort matrices by indexing any of its dimensions.

2.16.1 index

```
matrix = m:index(dim, idx)
```

This method returns a new deep cloned `matrix` but taking only the indexes of dimension `dim` indicated at `idx` object. The `idx` object can be three different data types:

- A one-dimensional `matrixInt32` with all the indices you want to take.

- A Lua `table` with the indices you want to keep. This table will be converted into a `matrixInt32` instance.
- A one-dimensional `matrixBool` were `true` value in the indices you want to keep.

This method can be combined with `eq`, `lt`, `gt` methods to select a bunch of rows or columns depending in a simple condition. The following example uses `index` and `lt` to select all the rows where its first column is less than 0.

```
> m = matrix(10,2):uniform(-10,10,random(1234))
> print(m)
 5             9
-4             2
 10            5
 7            -1
 1             2
...
-5            -8
# Matrix of size [10,2] stride [2,1] ref [0x204cce0 data= 0x1e3c310]
> m_neg_idx = m:select(2,1):lt(0)
> print(m_neg_idx)
F T F F F F F F F T
# MatrixBool of size [10] stride [1] ref [0x1f64820 data= 0x1e886a0]
> m_neg = m:index(1, m_neg_idx)
> print(m_neg)
-4             2
-5            -8
# Matrix of size [2,2] stride [2,1] ref [0x1d72450 data= 0x1f499a0]
```

2.16.2 indexed_fill

`m = m:indexed_fill(dim, idx, number)`

This method fills with `number` all the values at the given dimension `dim` whose indices are in `idx` object. The `idx` object can be three different data types:

- A one-dimensional `matrixInt32` with all the indices you want to take.
- A Lua `table` with the indices you want to keep. This table will be converted into a `matrixInt32` instance.
- A one-dimensional `matrixBool` were `true` value in the indices you want to keep.

The following example fills with 0 the values at column 1 which are negative.

```
> m = matrix(10,2):uniform(-10,10,random(1234))
> print(m)
 5             9
-4             2
 10            5
 7            -1
 1             2
...
-5            -8
# Matrix of size [10,2] stride [2,1] ref [0x204cce0 data= 0x1e3c310]
```

```
> m_neg = m:index(1, m:select(2,1):lt(0))
> m_neg_idx = m:select(2,1):lt(0)
> print(m_neg_idx)
F T F F F F F F F T
# MatrixBool of size [10] stride [1] ref [0x1f64820 data= 0x1e886a0]
> col1 = m(':',1)
> col1:indexed_fill(1, m_neg_idx, 0)
> print(m)
 5             9
 0             2
 10            5
 7            -1
 1             2
...
 0            -8
# Matrix of size [10,2] stride [2,1] ref [0x237cba0 data= 0x237cc70]
```

2.16.3 indexed_copy

`m = m:indexed_copy(dim, idx, matrix)`

This method copies the values of `matrix` at the indices of dimension `dim` indicated by `idx` object. The `idx` object can be three different data types:

- A one-dimensional `matrixInt32` with all the indices you want to take.
- A Lua `table` with the indices you want to keep. This table will be converted into a `matrixInt32` instance.
- A one-dimensional `matrixBool` were `true` value in the indices you want to keep.

2.16.4 masked_fill

`m:masked_fill(mask, value)`

2.16.5 masked_copy

`m:masked_copy(mask, value)`

2.16.6 order

`matrixInt32 = m:order()`

Returns a permutation of the caller `matrix` which sorts its data. The permutation is given as a `matrixInt32` with the indices of the caller `matrix`. The caller `matrix` shold be a one-dimensional matrix (rank 1 tensor).

```
> m = matrix(10,2):uniform(-10,10,random(1234))
> print(m)
 5             9
-4             2
 10            5
 7            -1
 1             2
```

```
...
-5            -8
# Matrix of size [10,2] stride [2,1] ref [0x10bf090 data= 0x10bf160]
> ord = m:select(2,1):order()
> print(ord)
         10           2           5           1 ...           3
# MatrixInt32 of size [10] stride [1] ref [0xe4eba0 data= 0xeb3230]
> sorted = m:index(1, ord)
> print(sorted)
-5            -8
-4             2
 1             2
 5             9
 5             8
...
 10            5
# Matrix of size [10,2] stride [2,1] ref [0x1c270c0 data= 0x1ef0080]
```

2.16.7 order_rank

```
matrixInt32 = m:order_rank()
```

Returns the rank of the `matrix` elements in its sorted permutation. The caller `matrix` should be a one-dimensional matrix (rank 1 tensor).

```
> m = matrix(10,2):uniform(-10,10,random(1234))
> print(m)
 5             9
-4             2
 10            5
 7            -1
 1             2
...
-5            -8
# Matrix of size [10,2] stride [2,1] ref [0x10bf090 data= 0x10bf160]
> rnk = m:select(2,1):order_rank()
> print(rnk)
          4           2          10           9 ...           1
# MatrixInt32 of size [10] stride [1] ref [0x2cf0100 data= 0x2cf0020]
```

2.17 Matrix class operations (no instance methods)

More matrix operations are located at `matrix.op`, most of them are equivalent to methods in `matrix` objects, but implemented to be **allocate new memory** instead of being in-place, and receiving the caller `matrix` as first argument. New operations have been defined in this table to reduce the overhead of `matrix` instance table. Only the new operations are documented here.

2.17.1 matrix.op.repmat

2.17.2 matrix.op.diag

2.17.3 matrix.op.triu

2.17.4 matrix.op.tril

2.18 Other Matrix operations (extensions)

The table `matrix.ext` contains new function extensions which work over matrices. This functions are here because the extensions are pretty new, still in testing, and to not polute too much the `matrix` class with new methods.

2.18.1 matrix.ext.real_fftwh

```
matrix = matrix.ext.real_fftwh(m, wsize=m:size(), wadvance=wsize, dest=nil)
```

This functions computes real FFT with Hamming window to the given matrix `m`. The FFT is computed for all the possible positions of a sliding window of size `wisize`. The sliding window advances `wadvance` samples every iteration. By default, the function configures `wsize` and `wadvance` just to compute FFT over the whole given matrix `m`. The matrix `m` should have only one dimension (rank 1). The result is new allocated matrix with size `NxF` where `N=(m:size() - wsize)/wadvance + 1` and `F` is the size of the FFT output. In case the argument `dest` is given, it should be a `matrix` instance of size `NxF`, and it would be the result `matrix`, avoiding the allocation of a new `matrix`.

```
> a = matrix(200):uniformf(0,1,random(1234))
> f = matrix.ext.real_fftwh(a, 20, 10)
> print(f)
 5.78204       17.9694       2.60205       1.93296      ...  0.210018
 4.80782       11.8858       0.97495       1.46241      ...  2.67692
 4.15355       9.48432       1.71497       0.480176     ...  0.498562
 5.12262       16.0089       3.65927       0.276271     ...  1.00497
 5.57981       16.7664       2.00985       0.0188873    ...  0.336096
...
 4.96313       15.267        3.44396       0.307427     ...  0.539195
# Matrix of size [19,16] stride [16,1] ref [0x1efbe50 data= 0x1f46330]
```

2.18.2 matrix.ext.iterate

2.18.3 matrix.ext.convolution

Chapter 3

Sparse matrix

`matrix.sparse`

3.1 Constructors

```
s = matrix.sparse(matrix)
```

```
s = matrix.sparse(d1, d2, values, indices, first_index)
```

```
s = matrix.sparse.csc(matrix)
```

```
s = matrix.sparse.csr(matrix)
```

3.1.1 diag

```
s = matrix.sparse.diag(obj, [format="csr"])
```

- A `number`
- A `matrix`
- A `table`

3.2 Dictionary of keys builder

```
dok = matrix.sparse.builders.dok()
```

3.2.1 set

```
dok = dok:set(row, col, value)
```

3.2.2 build

```
sparse = dok:build([num_rows, [num_cols, [format="csr"]]])
```

3.3 Sparse matrix Basic methods

3.3.1 size

3.3.2 non_zero_size

3.3.3 get

3.3.4 iterate

3.3.5 fill

3.3.6 zeros

3.3.7 ones

3.3.8 to_dense

3.3.9 get_sparse_format

3.3.10 dim

3.3.11 slice

3.3.12 clone

3.3.13 transpose

3.3.14 isfinite

3.3.15 min

3.3.16 max

3.3.17 equals

3.3.18 sqrt

3.3.19 pow

3.3.20 tan

3.3.21 tanh

3.3.22 atan

3.3.23 atanh

3.3.24 sin

3.3.25 sinh

3.3.26 asin

3.3.27 asinh

3.3.28 abs

Chapter 4

Other kind of matrices

Currently is possible to use complex, double, int32 and char matrices, supporting load and save, matrix structural methods, and some of them also support mathematical operations:

- `matrixBool`: a matrix of boolean values. Basic functionality.
- `matrixComplex`: fully working matrix type, with almost all the methods described above.
- `matrixDouble`: partial working matrix type, only allow structural methods (explained at MatFormat section).
- `matrixInt32`: partial working matrix type, only allow structural methods (explained at MatFormat section).
- `matrixChar`: partial working matrix type, only allow structural methods (explained at MatFormat section).

In all cases, you could use `april_help` to ask which methods are available. Complex, MatrixChar type implements a method `to_string_table`.

All matrices implement method `convert_to(type)` which receives a type string with one of these values: `"float", "double", "complex", "int32", "bool", "char"`; allowing to convert a matrix of one type into a different type with a casting and precision loss in some cases.

```
> m = matrix(5,5):uniformf(-10,10,random(12354))
> print(m)
 1.03926      -6.82017      -7.80579      -2.02109      -9.40496
 1.54086      -1.43963       7.11541       1.20382      -2.91477
 5.89334       9.04666      -0.688719     -9.25703      -3.0825
 5.4027       -3.25782      -0.7066       -5.90035      -8.35659
-0.986174      8.1467       -7.01133      -9.03494       5.69565
# Matrix of size [5,5] stride [5,1] ref [0x16d0410 data= 0x16d04e0]
> print(m:convert_to("int32"))
          1          -6          -7          -2          -9
          1          -1           7           1          -2
          5           9           0          -9          -3
          5          -3           0          -5          -8
          0           8          -7          -9           5
# MatrixInt32 of size [5,5] stride [5,1] ref [0x187d2b0 data= 0x187d380]
```

4.1 matrixComplex

The constructor of a `matrixComplex` receives a table with complex numbers (see utils section). A complex number uses float single precission resolution for real and imaginary part:

```
> -- using strings which are converted to complex numbers (slow performance)
> m = matrixComplex(2,2, { "1+1i", "2+2i", "3+2i", "4+1i" })
> = m
        1+1i         2+2i
        3+2i         4+1i
# MatrixComplex of size [2,2] in row_major [0x24d52c0 data= 0x24d4a00]
>
> -- using directly complex numbers
> m = matrixComplex(2,2, { complex(1,1), complex(2,2), complex(3,2), complex(4,1) })
> = m
        1+1i         2+2i
        3+2i         4+1i
# MatrixComplex of size [2,2] in row_major [0x24d6550 data= 0x24d6650]
```

Besides the standard `matrix` methods, the `matrixComplex` implements the following:

- `caller = m:conj()` computes the conjugate **in-place**, modifying the caller matrix, and returning the caller matrix instance.
- `matrix = m:real()` returns the real part of the caller `matrixComplex`.
- `matrix = m:img()` returns the imaginary part of the caller `matrixComplex`.
- `matrix = m:abs()` returns the modulus of the polar form of `matrixComplex`.
- `matrix = m:angle()` returns the angle of the polar form of `matrixComplex`.
- `matrix = m:to_float()` converts the caller matrix in a `matrix` object which has one additional dimension. This additional dimension has always size 2, and keeps the real and imaginary parts of the caller `matrixComplex`. The additional dimension will be the last. **Note** that the returned `matrix` and the `matrixComplex` caller references the same memory pointer.

4.2 matrixDouble

`matrixDouble` is the type of matrices for double data. This kind of matrices don't accept mathematical operations, but yes structural operations as `select`, `slice`, etc.

```
> m = matrixDouble(2,3,{1,2,3,4,5,6})
> = m
          1           2           3
          4           5           6
# MatrixDouble of size [2,3] [0x2512c70 data= 0x251ad70]
```

4.2.1 matrixInt32

`matrixInt32` is the type of matrices for integer data. This kind of matrices don't accept mathematical operations, but yes structural operations as `select`, `slice`, etc.

```
> m = matrixInt32(2,3,{1,2,3,4,5,6})
> = m
```

```
          1           2           3
          4           5           6
# MatrixInt32 of size [2,3] [0x2512c70 data= 0x251ad70]
```

4.3 matrixChar

`matrixChar` is the type of matrices for char data. This kind of matrices don't accept mathematical operations, but yes structural operations as `select`, `slice`, etc.

Exists an special method, `to_string_table()`, which converts the matrix in a table of strings, concatenating the chars in `row_major` order.

```
> m = matrixChar(2,2, { "h","ola" })
> = m
[1,1] = h
[1,2] = o
[2,1] = l
[2,2] = a
# MatrixChar of size [2,2] [0x12c3310 data= 0x12c3410]
> = unpack(m:to_string_table())
ho  la
```

Chapter 5

Matrix dictionary tools

The table `matrix.dict` contains several functions which allow to execute `matrix` operations over tables of matrices.

5.1 clone

```
another = matrix.dict.clone(tbl)
```

Returns a deep copy of the table.

5.2 clone_only_dims

```
another = matrix.dict.clone_only_dims(tbl)
```

Returns a deep copy of the table, **but** without copying the matrix data content, only cloning the matrix dimension sizes.

5.3 Implemented operations

The following list of operations are implemented to be executed over all the contained matrices:

- `number = matrix.dict.size()`
- `tbl = matrix.dict.fill(tbl,number)`
- `tbl = matrix.dict.ones(tbl)`
- `tbl = matrix.dict.zeros(tbl)`
- `tbl = matrix.dict.axpy(tbl, number, tbl2)`
- `number = matrix.dict.dot(tbl, tbl2)`
- `tbl = matrix.dict.copy(tbl, tbl2)`
- `tbl = matrix.dict.scalar_add(tbl, number)`
- `tbl = matrix.dict.complement(tbl)`

- `tbl = matrix.dict.pow(tbl, number)`
- `tbl = matrix.dict.scal(tbl, number)`
- `tbl = matrix.dict.inv(tbl)`
- `tbl = matrix.dict.sqrt(tbl)`
- `tbl = matrix.dict.exp(tbl)`
- `tbl = matrix.dict.plogp(tbl)`
- `tbl = matrix.dict.log1p(tbl)`
- `tbl = matrix.dict.cos(tbl)`
- `tbl = matrix.dict.cosh(tbl)`
- `tbl = matrix.dict.acos(tbl)`
- `tbl = matrix.dict.acosh(tbl)`
- `tbl = matrix.dict.tan(tbl)`
- `tbl = matrix.dict.tanh(tbl)`
- `tbl = matrix.dict.atan(tbl)`
- `tbl = matrix.dict.atanh(tbl)`
- `tbl = matrix.dict.sin(tbl)`
- `tbl = matrix.dict.sinh(tbl)`
- `tbl = matrix.dict.asin(tbl)`
- `tbl = matrix.dict.asinh(tbl)`

Chapter 6

tokens package

6.1 Introduction

Package `tokens` could be loaded via the standalone binary, or in Lua with `require("aprilann.tokens")`.

A `Token` is an abstract C++ class which has different specializations for different tasks and purposes. This class and its specializations are binded to Lua, allowing Lua scripts to be a glue language between C++ algorithms which are developed over the `Token` abstraction.

In this way, the current implementation of Artificial Neural Networks (ANNs) receives as input a token and produce a as output a token. Normally, for ANNs, this token contains a `matrix`, but it is also possible to wrap `matrix.sparse` objects and token vectors. Every ANN component checks the token type, and specialized computations could be done depending in the token type.

From Lua side, `matrix` instances can be given and taken as tokens.

6.2 Abstract class: `tokens.base`

This class defines the basic interface shared for all token types. It is an abstract class which couldn't be instantiated in Lua. If you try, you will see the following message:

```
> tokens.base()
tokens.base:tokens.base: Abstract class!!!
stack traceback:
    [C]: in function 'base'
    stdin:1: in main chunk
    [C]: in ?
```

Usually, C++ method calls return an instance to the class `tokens.base`, but in some cases it is possible to transform the generic reference to any of the child classes, or in other cases, it is possible to retrieve the contained data.

The transformation between tokens and matrix instances is performed automatically by the glue code between C++ and Lua.

6.2.1 Interface methods

The following methods are in the generic interface of `tokens.base` class.

6.2.1.1 clone

```
token = t:clone()
```

The method clone is implemented in all the tokens, and returns a **deep copy** of the caller.

6.3 Token types

6.3.1 Token bunch vector

```
tokens.vector.bunch
```

The token bunch vector is an array of tokens. Some ANN components allow to receive a vector of tokens which could contain sparse vectors.

```
> t = tokens.vector.bunch()
> t = tokens.vector.bunch(10)
> t = tokens.vector.bunch{ t1, t2, ... }
```

6.3.1.1 clear

```
t = t:clear()
```

6.3.1.2 size

```
number = t:size()
```

6.3.1.3 set

```
t = t:set(position, token)
```

```
> = t:set(1, matrix(2,3):linear())
```

6.3.1.4 push_back

```
t = t:push_back(token)
```

```
> = t:push_back(matrix(1,2):linear())
```

6.3.1.5 at

```
token = t:at(position)
```

```
> = t:at(1)
> = t:at(2)
```

6.3.1.6 iterate

```
lua iterator = t:iterate()
```

```
> for i,v in t:iterate() do print(i) print(v) end
```

Chapter 7

dataset package

7.1 Introduction

Package `dataset` could be loaded via the standalone binary, or in Lua with `require("aprilann.dataset")`.

The `dataset` table is a namespace and a Lua abstract class which adds an abstraction layer of set of patterns to the multi-dimensional matrices. It is also possible to do patterns pre-processing and, union and join operations of different datasets, an identity matrix dataset, and so on.

Every dataset implements following methods:

- `number = ds:numPatterns()`, it returns the number of patterns in the given `ds` dataset.
- `number = ds:patternSize()`, it returns the size of one pattern.
- `table = ds:getPattern(i)`, it receives a number between 1 and numPatterns(), and returns a table with the i-th pattern.
- `ds:putPattern(i,t)`, it receives a number between 1 and numPatterns(), and a table with patternSize() numbers, and overwrites the i-th pattern with the given table.
- `iterator = ds:patterns()`, an iterator function to use in Lua `for` statements: `for i,t in ds:patterns() do ... end`.
- `table = ds:mean()`, it returns the mean per each pattern component.
- `table,table = ds:mean_deviation()`, it returns the mean and standard deviation per each pattern component.
- `number,number = ds:min_max()`, it returns the minimum and maximum value of the dataset.
- `ds:normalize_mean_deviation()`, it receives two tables of patternSize length, the first with means, and the second with standard deviations, and the method normalizes the data substracting mean and dividing by standard deviation.
- `matrix = ds:toMatrix()`, it returns a new allocated bi-dimensional `matrix` object which contains all dataset patterns (numPatterns rows and patternSize columns).

7.2 dataset.matrix

This is the most important kind of dataset, allowing to create patterns moving a multi-dimensional window through a `matrix` object. This `dataset` takes the `matrix` by **reference**, so any change in the `matrix` will be reflected in the patterns produced by the `dataset`:

```
xor_in = matrix(4,2, {0,0,
                      0,1,
```

```
                      1,0,
                      1,1})
xor_out = matrix(4, {0, 1, 1, 0})
-- by default, dataset.matrix traverses the matrix by rows
ds_xor_in  = dataset.matrix(xor_in)
ds_xor_out = dataset.matrix(xor_out)
```

For a given matrix with dimensions n1,n2,...,nk, by default the dataset contains n1 number of patterns with size n2 x ... x nk. For a bidimensional matrix it is a row-major order traversal. For a vector, it is the traversal of all its elements:

```
> a = matrix(2, 2, {1,2,3,4})
> b = dataset.matrix(a)
> for i,j in b:patterns() do print(table.concat(j,",")) end
1,2
3,4
> a = matrix(2,2,2,{1,2,3,4,5,6,7,8})
> b = dataset.matrix(a)
> for i,j in b:patterns() do print(table.concat(j,",")) end
1,2,3,4
5,6,7,8
> a = matrix(4,{1,2,3,4})
> b = dataset.matrix(a)
> for i,j in b:patterns() do print(table.concat(j,",")) end
1
2
3
4
```

Until this point, none benefit of `dataset` over `matrix` is presented. We are going to show that for the same given `matrix`, we could generate several different `dataset` modifying some parameters which has been taken by default until now.

When we instantiate a `dataset.matrix`, the first argument is a **K**-dimensional `matrix` with size `n1 X n2 x ... x nK`. The second argument could be a Lua table with the following fields:

- `patternSize`, a table array with **K** positive integers. It indicates the size of each pattern taken from the underlying `matrix`. By default it is `patternSize={ 1, n2, n3, ..., nK }`.
- `offset`, a table array with **K** signed integers. It indicates the offset of the first pattern. A negative value is useful to compute a pattern which traverses the `matrix` limits. The first initial position is `0`. Its default value is `offset={ 0, 0, ..., 0 }`.
- `numSteps`, a table with **K** estrict positive integers (> 0). It indicates the number of steps used for each dimension to generate all the possible patterns. Its default value is `numSteps={ n1, 1, ..., 1 }`. The total `numPatterns()` method returns the product of all `numSteps` components.
- `stepSize`, a table with **K** signed integers. It indicates the number of coodinates which are slided for each dimension with every pattern. Its default value is `stepSize={ 1, ..., 1 }`. Obviusly, in every `i` dimension where `numSteps[i]=1`, the `stepSize[i]` is not important. Depending on the values of `stepSize` and `patternSize`, the `matrix` will be traversed with overlapping between patterns or not.
- `orderStep`, a table with a permutation of the **K** dimensions, indicating the order for `matrix` traversal. By default, the `matrix` is traversed in `row_major` order, so its value is `orderStep={ K-1, K-2, ..., 2, 1, 0 }`. Varying the order of this numbers, it is possible to produce a different order traversal, as for example a `col_major` order.

- `defaultValue` is a number (not necesarily an integer), used to fill the pattern positions which are out of the `matrix` limits. By default its value is `defaultValue=0`.
- `circular` is a table with **K** booleans (true or false) which indicate for every `matrix` dimension if it is circular or not. By default it is false in all dimensions `circular={ false, false, ..., false }`. When a dimension is not circular, the pattern positions out of the `matrix` limits are filled with `defaultValue`. When a dimension is circular, the pattern positions out of the `matrix` are re-interpreted starting at the first position of this dimension in the matrix. For example, a bi-dimensional `matrix` whith one circular dimension seems cilindrical. If the two dimensions are circular, it seems thyroidal (like a donut).

Look a short example of this parameters. We want to generate a dataset with binary XOR patterns using only one `matrix`:

```
> m_xor = matrix.fromString[[
4 3
ascii
0 0 0
0 1 1
1 0 1
1 1 0
]]
> ds_input  = dataset.matrix(m_xor,{patternSize={1,2}})
> ds_output = dataset.matrix(m_xor,{offset={0,2},patternSize={1,1}})
> for i=1,ds_input:numPatterns() do
>> printf("%d -> Input: %s Output: %s\n",i,
>> table.concat(ds_input:getPattern(i),","),table.concat(ds_output:getPattern(i),","))
>> end
1 -> Input: 0,0 Output: 0
2 -> Input: 0,1 Output: 1
3 -> Input: 1,0 Output: 1
4 -> Input: 1,1 Output: 0
```

We could implement the following function:

```
function dataset_pair(m,sizein,sizeout)
  local d_in  = dataset.matrix(m,{patternSize = {1,sizein}})
  local d_out = dataset.matrix(m,{offset={0,sizein},patternSize = {1,sizeout}})
  return d_in,d_out
end
-- which could be used as this
ds_input,ds_output = dataset_pair(m_xor,2,1)
```

7.3 dataset.identity

This `dataset` represents the traversing of an identity `matrix`. It receives as first argument the number of patterns (which is at the same time the patternSize), a second **optional** argument which is the value of zero (by default is `0.0`), and a third **optional** argument with the value of one (default is `1.0`).

```
> ds_eye = dataset.identity(5)
> print(ds_eye:toMatrix())
```

```
 1           0           0           0           0
 0           1           0           0           0
 0           0           1           0           0
 0           0           0           1           0
 0           0           0           0           1
# Matrix of size [5,5] in row_major [0x1418bd0 data= 0x1418cd0]
```

The `dataset.identity` is equivalent to following code, but is more efficient:

```
> ds_eye = dataset.matrix(matrix(5,5):zeros():diag(1))
> print(ds_eye:toMatrix())
 1           0           0           0           0
 0           1           0           0           0
 0           0           1           0           0
 0           0           0           1           0
 0           0           0           0           1
# Matrix of size [5,5] in row_major [0x129f930 data= 0x12fb470]
```

7.4 dataset.indexed

The `dataset.indexed` allows to map indexes with patterns. It is useful to specify the output of a classification task, in which case the underlying `dataset` will be the association of ANN output for each of the classes. Another possibility is to use `dataset.indexed` to select a random set of patterns from the underlying `dataset`. **NOTE** that `dataset.indexed` uses float numbers to represent the indices, so the maximum integer number which could be indexed is `16777216`. If you need more resolution, use `dataset.index_filter` (which is less general than this).

The constructor receives 2 arguments, the first is the base `dataset`. The second is a table array with as many `dataset` objects as `patternSize()` of the base `dataset`, acting every one of this as a dictionary. The `patternSize()` of the resulting `dataset.indexed` object is equals to the sum of the `patternSize()` of all the dictionaries.

Following code is an example for a classification task ANN output:

```
> dict = dataset.identity(10)
> -- a random matrix with integers [1,10]
> m_base = matrix(100):uniform(1,10,random(1234))
> ds_base = dataset.matrix(m_base)
> indexed_ds = dataset.indexed( ds_base, { dict })
```

The following is code for a random subset of patterns from a given `dataset`:

```
-- a matrix with 100 patterns with real numbers in [-1,1]
> m_dict = matrix(100, 10):uniformf(-1,1,random(1234))
> dict = dataset.matrix(m_dict)
> -- a random matrix with 10 integers in range [1,100], a selection of patterns
> m_base = matrix(10):uniform(1,100,random(1234))
> ds_base = dataset.matrix(m_base)
> indexed_ds = dataset.indexed( ds_base, { dict })
```

7.5 dataset.index_filter

The `dataset.index_filter` is like `dataset.indexed` but only for the case of indexing a random subset of patterns from a given base `dataset`, which receives as first argument. As second argument, a vector of unsigned integers (`util.vector_uint`) is expected.

```
> -- a dataset with 100 patterns of size 5, randomized at range [0,1]
> base_ds = dataset.matrix(matrix(100,5):uniformf())
> uint_vector = util.vector_uint()
> rnd = random(1234)
> -- a subset of 10 patterns from indices at range [1,100]
> for i=1,10 do uint_vector:push_back( rnd:randInt(1,100) ) end
> print(uint_vector)
      48       84       39       54       77       25       16       50
      24       27
# vector_uint of size 10
> index_filter_ds = dataset.index_filter(base_ds, uint_vector)
> print(index_filter_ds:toMatrix())
 0.528819    0.915766    0.220549    0.828223    0.28173
 0.73919     0.424762    0.354582    0.368474    0.0355779
 0.512678    0.494687    0.731773    0.672073    0.411915
 0.575729    0.169612    0.346667    0.925921    0.332662
 0.298257    0.460495    0.179573    0.32725     0.610076
 0.219746    0.15807     0.581498    0.531874    0.200707
 0.00641197  0.86275     0.407079    0.279832    0.602674
 0.456097    0.463612    0.521626    0.951389    0.659111
 0.4136      0.734821    0.212726    0.314356    0.50499
 0.662668    0.584882    0.457253    0.325801    0.217475
# Matrix of size [10,5] in row_major [0x12a2710 data= 0x13eaa10]
```

7.6 dataset.join

The `dataset.join` object *joins* the outputs from several `dataset` objects which has the same `numPatterns`. The `patternSize` of the resulting `dataset` is equals to the sum of every `patternSize` of its components. It requieres as argument a table with the `datasets` which you want to join.

```
-- ds1, ds2 and ds3 are three datasets with the same numPatterns
> join_ds = dataset.join{ ds1, ds2, ds3 }
```

7.7 dataset.union

This `dataset` allows to convert several `dataset` objects with the same `patternSize` as they were one unique `dataset` which its `numPatterns` is equals to the sum of all the `numPatterns` of every given dataset. It receives only one argument, a table with the `dataset` which will be unionized.

```
> -- ds1, ds2 and ds3 are datasets with the same patternSize
> union_ds = dataset.union{ ds1, ds2, ds3 }
```

7.8 dataset.slice

The `dataset.slice` is useful to extract a contiguous subset of patterns from a given `dataset` (for more general subsets use `dataset.indexed` or `dataset.index_filter`). It requieres 3 arguments. The first is the base `dataset`. The second and third arguments are the initial and final indices of the patterns which form the subset (first valid index is 1, and last valid index is numPatterns() of base `dataset`).

```
> -- slice with 100 patterns, from 101 to 200
> slice_ds = dataset.slice(base_ds, 101, 200)
```

7.9 dataset.deriv

The `dataset.deriv` receives a dataset and outputs the original data, the first derivative, or the second derivative, depending on the parameters received. It receives a table with a maximum of four fields:

- `dataset`: the base dataset, which contains data for derivative computation.
- `deriv0`: an **optinal** boolean, by default is `true`, which indicates if the output of the dataset will contain the *original pattern*, without derivative.
- `deriv1`: an **optinal** boolean, by default is `true`, which indicates if the output of the dataset will contain the *first derivative*.
- `deriv2`: an **optinal** boolean, by default is `true`, which indicates if the output of the dataset will contain the *second derivative*.

```
> -- ds is the base dataset
> only_first_deriv_ds = dataset.deriv{ dataset=ds, deriv0=false, deriv1=true, deriv2=false }
```

7.10 dataset.contextualizer

The `contextualizer` is a `dataset` which adds context from the adjacent patterns (left and right). If any of the adjacent patterns is out of the base `dataset` size, it fills it with the first or the last pattern. The constructor receives four arguments:

1. The base `dataset`.
2. The size of the left context.
3. The size of the right context.
4. A `boolean` optionally argument indicating if the left and right contexts needs to be swapped. By default is `false`, and in almost all cases it is what you need ;)

```
> ds = dataset.contextualizer(dataset.identity(2,0,1),1,1)
>
> print(ds:toMatrix())
 1           0           1           0           0           1
 1           0           0           1           0           1
# Matrix of size [2,6] in row_major [0x18357b0 data= 0x18358b0]
```

7.11 dataset.split

This `dataset` allows to select a subset of the components of patterns produced by another `dataset`. So, the resulting `dataset` will have the same number of patterns, but different pattern size. The subset is an interval of the base `dataset`. It receives three positional arguments:

1. The base `dataset`.
2. The first position in the interval (counting from 1).
3. The last position in the interval (counting from 1).

```
> ds = dataset.split(dataset.identity(5,0,1), 2, 4)
> print(ds:toMatrix())
 0           0           0
 1           0           0
 0           1           0
 0           0           1
 0           0           0
# Matrix of size [5,3] in row_major [0xcb0f80 data= 0xcb1080]
```

7.12 dataset.perturbation

7.13 dataset.salt__noise

7.14 dataset.sub__and__div__normalization

This `dataset` applies *on-the-fly* a subtraction and division normalization, as for example a zero-mean one-standard-deviation normalization. So, for a `dataset` with `N` patternSize, given a vector of sub values `s1, s2, ..., sN`, and a vector of div values `d1, d2, ..., dN`, a `ds:getPattern(i)` of the resulting `dataset` will produce a pattern with `(v1-s1)/d1, (v2-s2)/d2, ..., (vN-sN)/dN`, being `vj` the `j` component of pattern `i`.

```
> eye_ds = dataset.identity(5,0,1)
> sub,div = {1,2,-1,2,-1},{0.1,0.1,0.1,0.1,0.1}
> ds = dataset.sub_and_div_normalization(eye_ds,sub,div)
> print(ds:toMatrix())
 0          -20          10         -20          10
-10         -10          10         -20          10
-10         -20          20         -20          10
-10         -20          10         -10          10
-10         -20          10         -20          20
# Matrix of size [5,5] in row_major [0xf47d70 data= 0xcfa060]
```

Chapter 8

The token dataset: dataset.token

8.1 Methods

8.1.1 numPatterns

8.1.2 patternSize

8.1.3 getPattern

```
token = ds:getPattern(number)
```

8.1.4 getPatternBunch

```
token = ds:getPatternBunch(table)
```

8.1.5 putPattern

8.1.6 putPatternBunch

8.1.7 patterns

8.2 dataset.token.sparse_matrix

```
ds = dataset.token.sparse_matrix(sparse matrix in CSR)
```

```
> m = matrix.sparse.diag{1,2,3,4,5,6}
> ds = dataset.token.sparse_matrix(m)
> print(ds:getPattern(1))
 1           0           0           0           0           0
# SparseMatrix of size [1,6] in csr [0x2aea350 data= 0x2aea420 0x2aea4a0 0x2aea4e0], 1 non-zeros
> print(ds:getPatternBunch{3,5})
 0           0           3           0           0           0
 0           0           0           0           5           0
# SparseMatrix of size [2,6] in csr [0x2aeab70 data= 0x2aea4a0 0x2aea420 0x2aea7b0], 2 non-zeros
```

8.3 dataset.token.union

```
ds = dataset.token.union(table)
```

8.4 dataset.token.vector

```
ds = dataset.token.vector(psize)
```

```
ds:push_back(token)
```

8.5 dataset.token.filter

```
ds = dataset.token.filter(dataset, obj)
```

8.6 My own Lua dataset.token

It is possible to develop Lua `dataset` classes which has to complain interface of `dataset.token` class. The unique restriction is that your Lua `dataset` couldn't be used as input to other C++ `dataset` objects. However, the Lua `dataset` can use C++ objects or Lua objects without making any distinction.

The following is a piece of a pure Lua `dataset.token` which replicates the behavior of `dataset.join`, but using tokens. `matrix` type is needed for instances which you want to join.

```
ds_join,ds_join_methods = class("ds_join")

function ds_join:constructor(t)
  assert(type(t)=="table" and #t>0,
         "Needs an array of dataset.token instances as argument")
  local psize = 0  -- we sum here the pattern size of all the given datasets
  local nump  = 0  -- we store here the number of patterns, which must be
                   -- equals in all the given datasets
  local data  = {} -- this table will store the given datasets
  for _,v in ipairs(t) do
    psize = psize + v:patternSize()
    local aux_nump = v:numPatterns()
    assert(nump==0 or nump==aux_nump)
    nump = aux_nump
    table.insert(data, v)
  end
  self.data=data
  self.num_patterns=nump
  self.pattern_size=psize
end

function ds_join_methods:numPatterns() return self.num_patterns end

function ds_join_methods:patternSize() return self.pattern_size end

function ds_join_methods:getPattern(idx)
  -- use the given matrix or construct a new one
```

```
  local m = matrix(1,self:patternSize())
  local col_pos = 1
  for _,ds in ipairs(self.data) do
    local psize  = ds:patternSize()
    local dest_m = m:slice({1,col_pos}, {1,psize})
    dest_m:copy(ds:getPattern(idx))
    col_pos = col_pos + psize
  end
  return m
end

function ds_join_methods:getPatternBunch(idxs)
  -- use the given matrix or construct a new one
  local m = matrix(#idxs,self:patternSize())
  assert(m:dim(1)==#idxs and m:dim(2)==self:patternSize())
  local col_pos = 1
  for _,ds in ipairs(self.data) do
    local psize  = ds:patternSize()
    local dest_m = m:slice({1,col_pos}, {#idxs,psize})
    dest_m:copy(ds:getPatternBunch(idxs))
    col_pos = col_pos + psize
  end
  return m
end
```

Chapter 9

ann package

9.1 Introduction

Several packages contain neural networks stuff: `require("aprilann.ann")`, `require("aprilann.ann.loss")`, `require("aprilann.ann.optimizer")`, `require("aprilann.trainable")`.

This page describe the utilities to build and train ANNs. Four main sections are written: a desciprion of ANN concepts in APRIL-ANN, the easy building procedure for MLPs, the training helpers, and finally the full description of the `aprilann.ann` package.

9.2 ANN components

Inspired by other toolkits (as Torch 7 or pyBrain), ANNs are described as a composition of blocks call ANN components, so one component is a neural network itself. A list of all available components appears executing:

```
april_help(ann.components)
```

Nevertheless, the composition procedure will be explained later. An ANN component is identified by a name string (which will be automatically generated if not given). The name must be unique. Some components contains weights in their core, which are estimated by gradient descent algorithm (backpropagation). Connection weights objects are identified by a weights name parameter, which could be reused. If two components have the **same** weights name, then they **share** the same connections object.

All components have an input and output size, which defines the number of weights (if needed) and the fan-in/fan-out of the component. Components need to be build (build method) once they are constructed. Build procedure allocates memory for connections and checks input/output sizes of components.

More accurate description is available at `april_help`, but don't be affraid, the next section presents an abstraction for train MLPs which automatically does a lot of this work:

```
april_help(ann.components.base)
april_help(ann.components.base.build)
```

9.3 The easy way: all-all MLP

The simpliest kind of ANN is a Multilayer Perceptron (MLP) where each layer is fully connected with the next layer (feed-forward, all-all connections).

9.3.1 Building the MLP: ann.mlp.all_all.generate

The method `generate` returns an special component object, which cannot be modified. Actually, it is a Lua table formed by an `ann.components.stack` instance and other information useful to load and save the MLPs, and it implements wrapper Lua functions to ANN component methods.

```
-- creates an ANN component for a MLP with the given description
thenet = ann.mlp.all_all.generate("256 inputs 128 tanh 10 log_softmax")

-- creates an instance of a trainer object for previous ANN component,
-- using the multi-class cross-entropy loss function (for 10 output units),
-- and using a bunch_size of 32. Loss function and bunch_size are optional.
trainer = trainable.supervised_trainer(thenet,
                       ann.loss.multi_class_cross_entropy(10),
                       32,
                       -- this last parameter is optional, by default is
                       -- SGD => Stochastig Gradient Descent
                       ann.optimizer.sgd())

-- builds the component contained into trainer object
trainer:build()

-- initializes the weights randomly, using fan-in and fan-out
trainer:randomize_weights{
  random      = random(1234),
  inf         = -0.1,
  sup         =  0.1,
  use_fanin   = true,
  use_fanout  = true,
}
```

As said before, each component has a unique name, and if needed a weights name. The next code iterates over all components:

```
> for name,c in trainer:iterate_components() do print(name,c) end
actf1   instance 0x7fc3e94850a0 of ann.components.base
actf2   instance 0x7fc3e9485550 of ann.components.base
b1  instance 0x7fc3e9484f80 of ann.components.base
b2  instance 0x7fc3e9485410 of ann.components.base
c1  instance 0x7fc3e9484a10 of ann.components.base
layer1  instance 0x7fc3e9484e80 of ann.components.base
layer2  instance 0x7fc3e9485310 of ann.components.base
w1  instance 0x7fc3e9484ee0 of ann.components.base
w2  instance 0x7fc3e9485370 of ann.components.base
```

The MLP is composed by 9 components, two activation functions (actf1 and actf2), two bias components (b1 and b2), one stack component which works as a container (c1), two hyperplane components containing one bias and one dot_product each one (layer1 and layer2), and finally two dot_product components (w1 and w2) which contains weight matrixes.

It is also possible to iterate over all weigths names:

```
> for name,connections in trainer:iterate_weights() do print(name,type(connections)) end
b1  matrix
```

```
b2  matrix
w1  matrix
w2  matrix
```

So, our MLP contains two bias vectors (b1 and b2, corresponding with b1 and b2 components), and two weights matrixes (w1 and w2, corresponding with w1 and w2 components). All MLPs generated automatically assign this names to its components and weights.

One time the component is build by using a trainer instance, the trainer exposes two interesting methods `trainer:component(COMPONENT_NAME_STRING)` which returns the component given its name, and `trainer:weights(WEIGTHS_NAME_STRING)` which returns the connection weigths object given its weigths_name attribute.

More info about `trainable.supervised_trainer` doing:

```
april_help(trainable.supervised_trainer)
```

9.3.2 Loss functions: ann.loss

The loss function is used to train the ANNs via gradient descent algorithm. Trainer objects needs an instance of a loss function to perform training, being a very useful abstraction of standard training procedures.

Detailed information about loss functions is in:

```
april_help(ann.loss)
```

The loss function could be set at trainer constructor, or using the method set_loss_function:

```
trainer:set_loss_function(ann.loss.mse())
```

Three main error functions are implemented: mean square error (MSE), two class cross-entropy, and multi-class cross-entropy. Note that cross-entropy like functions are specialized for log_logistic or log_softmax output activation functions. Almost all the constructors accepts a `SIZE=0` parameter, which means that the layer has a dynamic size.:

- `ann.loss.mse(SIZE)` returns an instance of the Mean Squared Error error function for SIZE neurons. It is a quadratic loss function.
- `ann.loss.mae(SIZE)` returns an instance of the Mean Absolute Error function, for SIZE neurons. It is not a quadratic loss function.
- `ann.loss.cross_entropy(SIZE)` returns an instance of the two-class cross-entropy. It only works with `log_logistic` output activation function. It is based on Kullback-Leibler divergence.
- `ann.loss.multi_class_cross_entropy(SIZE)` returns an instance of the multi-class cross-entropy. The parameter must be `SIZE>2`, so for two-class problems only one output unit with cross-entropy is needed. It only works with `log_logistic` or `log_softmax` output activation function (its better to use `log_softmax`). It is based on Kullback-Leibler divergence.

9.3.3 ann.optimizer

The optimizer is an object which implements the learning algorithm. Every class in `ann.optimizer` is an optimizer. Several learning hyperparameters are available, depending in the selected optimizer. This learning hyperparameters are known as *options*, and could be set **globally** (to all the connection weight layers of the ANN), or **layerwise** (to a concrete connection weights object, identified by its name). Optimizers implement the following API:

- `other = optimizer:clone()`: returns a deep copy of the caller object.
- `value = optimizer:get_option(name)`: return the *global* value of a given learning option name.
- `optimizer:set_option(name, value)`: sets the *global* value of a given learning option name.
- `optimizer:set_layerwise_option(layer_name, option_name, value)`: sets a *layerwise* option.
- `value = optimizer:get_layerwise_option(layer_name, option_name)`: returns the *layerwise* option of the given.
- `value = optimizer:get_option_of(layer_name, option_name)`: returns the option which is applicable to the given `layer_name`. If a *layerwise* option was previously defined, the method returns its value. Otherwise, the value of the *global* option will be returned.

9.3.3.1 ann.optimizer.sgd

Different `optimizer` objects are implemented. They train the neural network following different algorithms which rely in the computation of gradients done by ANN components. Them incorporate regularization and momentum hyperparameters. They options are algorithm dependententendt. In case of *Stochastic Gradient Descent*, the options are:

- `learning_rate`: the learning rate controls the portion of the gradient used to update the weights. This value is smoothed depending in the `bunch_size` and in the number `K` of times that a weight connections object is shared between different components. The smoothing value: `learning_rate/sqrt(bunch_size+K)`
- `momentum`: is a inertial hyperparameter which applies a portion of the weight update in the previous iteration.
- `weight_decay`: a L2 regularization term.
- `L1_norm`: a L1 regularization term.
- `max_norm_penalty`: a constrain penalty based on the two-norm of the weights.

The algorithm uses the following learning rule:

```
w = (1 - weight_decay)*w' + momentum*(w' - w") + lr'*grad(L)/grad(w')
```

where `w`, `w'` and `w"` are the weight values at next, current, and previous iterations; `lr'` is the `learning_rate` smoothed by the `sqrt`, and `grad(L)/grad(w')` is the loss function gradient at the given weight.

9.3.4 Trainer set and get of hyperparameters

The hyperparemters of `optimizer` objects can be modified by the `trainer` object:

- `trainer:set_option(name,value)`: sets a global learning option value.
- `value=trainer:get_option(name)`: gets a global learning option value.
- `trainer:set_layerwise_option(layer_name_match,option_name,value)`: sets a layerwise learning option value of all the connection weight objects whose name *matches* the given `layer_name_match` Lua pattern string.
- `value=trainer:get_option_of(layer_name,option_name)`: gets the option value applicable to the given layer.

```
trainer:build()
trainer:set_option("learning_rate", number)
trainer:set_option("momentum", number)
-- regularization is recommended to not be applied at bias connections
trainer:set_layerwise_option("w.*", "weight_decay", number)
trainer:set_layerwise_option("w.*", "max_norm_penalty", number)

-- for dropout (see dropout http://www.cs.toronto.edu/~nitish/msc_thesis.pdf)

-- dropout is a very especial option, it modifies training, but also modifies
-- validation (or test) phase. Also it must be applied carefully to not apply
-- dropout at the output of your model. Dropout is applied as another component
-- which acts as a stochastic filter.
```

9.4 Supervised trainer description

See the documentation for `trainable` package.

9.4.1 Stopping criteria

See the documentation for `trainable` package.

9.5 ann package reference

ANNs are implemented as a composition of components which implements define the three main operations of an ANN: forward step (compute outputs), backprop step (neuron gradient computation), and gradient computation step (weight gradients). All components are child classes of `ann.components.base`. See `april_help(ann.components.base)` for on-line documentation.

Two main remarks before continue following sections. The components has two special properties:

- name: is a string which identifies the component in a unique manner, is forbidden that two components sharing the same name.
- weights_name: is a string which identifies the connections (weights or biases) of the component. This name could be share by different components, which means that they **share** the same connections object.

9.5.1 Tokens and matrices

The components are integrated in Lua via the abstract class `token`, which has two specializations for ANNs:

- `tokens.matrix` is a token which contains a `matrix` instance.
- `tokens.sparse_matrix` is a token which contains a `matrix.sparse` instance.

In any case, ANN components wrap the given `matrix` objects into a `token`, and unwrap `matrix` objects when returning a `token`. So, in practice, you can ignore the token/matrix association.

NOTE that ANN components work with dense matrix or with `csr` sparse matrices.

9.5.2 Components basis

All components has defined the following basic properties, which are tokens: input, output, error_input, and error_output. Four are the basic methods to train the components:

- `component,table,table = build()`: this method reserves memory for weights and prepares the component to work with.
- `reset(iteration)`: it releases all the tokens internally allocated (or given by Lua), and receives the current iteration number. This iteration is not related with the training loop or epoch, it is related to `optimizer` objects which implement line search or similar (Conjugate Gradient or RProp).
- `token=forward(token[, boolean])`: it receives an input token and returns the output token. For simplicity, it is possible to give a `matrix` instead of a `token`, and the method will wrap automatically the given `matrix`. In any case, the returned value is a `token`.
- `token=backprop(token)`: it receives an error input token (gradient), and returns the output error token (gradient). For simplicity, it is possible to give a `matrix` instead of a `token`, and the method will wrap automatically the given `matrix`. In any case, the returned value is a `token`.
- `gradients=compute_gradients( [gradients] )`: compute the weight gradients, by using the data stored at the components (input/output tokens, input/output error tokens), given and produced during `forward` and `backprop` methods. Additionally, it receives a table of `matrix` with previously computed gradients, which will be used to store the data avoiding the allocation of new memory. The method returns a table of `matrix` with the gradients computed for each connection weights object.

Combining this methods with loss functions a component could be trained following this basic example. A linear component is trained to follow OR function, for input=[0,1] and target output=[1]. By default the weights are not initialized, so they contains memory trash.

```
> o = ann.optimizer.gsd() -- the optimizer
> l = ann.loss.mse(1) -- MSE loss function
> -- an hyperplane component (explained later)
> c = ann.components.hyperplane{ input=2, output=1 }
> c:build() -- allocates memory for weights, and checks components integrity
> l:reset() -- set to zero all the things
> c:reset() -- set to zero all the things
> o:execute(function()
              -- the true indicates training
              output_token=c:forward(matrix(1,2,{0,1}), true)
              -- gradient with desired output 1
```

```
            output_error=c:backprop(l:gradient(output_token,
                                               matrix(1,1,{1})))
            grad = c:compute_gradients(grad)
            return l:compute_loss(output_token,
                                  matrix(1,1,{1}),
                   grad
          end, c:copy_weights())
> output_token=c:forward(matrix(1,2,{0,1}))
> print(output_token) -- the output is closer to 1
0.2
# Matrix of size [1,1] [0xb01ce0 data= 0xad97d0]
```

9.5.3 Methods common to all the components

Note that all matrices must had at least two dimensions. All computations are done in bunch mode (using mini-batches) and the first dimension size is the number of patterns contained by the bunch. The rest of dimensions must complain the input constrains of the component. A lot of components work with linear inputs, so the input matrix will be bi-dimensional, but some components work with multidimensional matrices. *It is possible to use matrices of only one dimension and they will be reinterpreted as two dimensional matrices with only one row, but better if you work always with two-dimensional matrices.*

9.5.3.1 Building procedure

Before doing anything, components could be composed together to build larger components. This procedure needs to call `build` method at the end, to check the input/output sizes and reserve memory for weights and biases.

The `c:build()` call executes recursively the build method of all the components composition. This method returns two tables:

```
> caller_component, weights_dict, components_table = c:build()
```

The `caller_component` is the component `c` in this case.

The `weights_dict` is a table of matrices, which indexes name (weight name) strings with weight matrices.

The `components_table` is a Lua table indexed by each name (component name) and contains a reference to the component instance, which is useful to initialize hyper-parameter and other stuff in a component-wise manner.

9.5.3.2 Input/output sizes

- `number = c:get_input_size()`: returns the size of the input for the caller component. In case of unknown input size, a zero will be returned.
- `number = c:get_output_size()`: returns the size of the output for the caller component. In case of unknown output size, a zero will be returned.
- `table = c:precompute_output_size( [table] )`: allows to compute the output size shape, given an input shape. It is useful to be combined with convolutional ANNs, in order to ask for the output shape size of the convolution. The given `table` must complains the expected input shape of the component (normally is one dimension, but with CNNs it could be multi-dimensional). The returned `table` will contain as many dimensions as the produced by the caller component (idem as for input).

9.5.3.3 Back-propagation computation methods

- `token = c:forward( token [, boolean] )` receives a token and an optional boolean (by default false). The boolean indicates if this forward is during training or not, because some components has an special behavior during training. It returns a token with the output computation of the caller component. For simplicity, it is possible to give a `matrix` instead of a `token`, and the method will wrap automatically the given `matrix`. In any case, the returned value is a `token`.
- `token = c:backprop( token )` receives a token with the input error (gradient of each output neuron), and returns another token with the output error (gradient of each input neuron). For simplicity, it is possible to give a `matrix` instead of a `token`, and the method will wrap automatically the given `matrix`. In any case, the returned value is a `token`.
- `gradients = c:compute_gradients( gradients )` returns the weight gradients computed using the tokens given at `forward` and `backprop` methods.
- `c:reset()` releases the retained tokens in `forward` and `backprop` steps.

9.5.3.4 Getters of produced and retained tokens

During `forward` and `backprop` steps the components compute outputs and error outputs (gradients), and retain the input and error input (gradients) tokens. Before call `reset` method, you could ask the component for its retained tokens:

- `token = c:get_input()` returns the token given as input at `forward` method.
- `token = c:get_output()` returns the token computed as output by `forward` method.
- `token = c:get_error_input()` retruns the token given as error input at `backprop` method.
- `token = c:get_error_output()` returns the token computed as error output by `backprop` method.

9.5.4 Weights matrices and bias vectors

Components which require weights has internally a `matrix` instance. This object is allocated calling the build method of the components (or using the build method of a trainer), and is identified by the weigths_name property, so components with the **same** weigths_name **share** the same connections object.

This matrices are defined with OUTPUTxINPUT size (output rows, input columns), so:

- Bias vectors: has INPUT=1 and OUTPUT=number of neurons, and they are a column vector.
- Weight matrices: contain OUTPUTSxINPUTS weights.

The weights matrices has this format:

```
w(i1,o1)  w(i2,o1)  w(i3,o1)  ...
w(i1,o2)  w(i2,o2)  w(i3,o2)  ...
...       ...       ...
```

where `w(a,b)` is the weight which connects input `a` with output `b`. Be sure that your matrices has this format.

9.6 Components list

The ANN models are modular components which can be sorted in several ways to produce different topologies.

9.6.1 Basic components

9.6.1.1 base

`ann.components.base{ size=0, [name=STRING] }`

The class `ann.components.base` is the base of all ANN components. It is possible to instance an object of this class, and it performs identity function. The constructor receives optionally the `name` of the component. The constructor receives two optional arguments, the `size=0`, by default it allows any input size, and the `name` of the component.

```
> c1 = ann.components.base{ name="base1" }
> c2 = ann.components.base()
> input = matrix(10,10):uniformf(0,1,random(237))
> output = c2:forward(input)
> = output:equals(input)
true
```

9.6.1.2 bias

`ann.components.bias{ size=NUMBER, [name=STRING], [weights=STRING] }`

The class `ann.components.bias` implements an additive bias of a given size. The bias is added iteratively to all the patterns in the bunch (mini-batch). The constructor receives two fields:

- `name` of the component, an **optional** field.
- `weights` name of the component, an **optional** field.
- `size` the size of the bias vector.

This components contains a vector of `SIZEx1`, which is added transposed to all the input patterns (first dimension of the bunch).

```
> b1 = ann.components.bias{ name='b1', weights='b1', size=5 }
> _,weights = b1:build()
> weights('b1'):linspace()
> = weights('b1')
 1
 2
 3
 4
 5
# Matrix of size [5,1] [0x162eb00 data= 0x16b0260]
> input = matrix(4,5):linspace()
> = input
 1          2          3          4          5
 6          7          8          9          10
 11         12         13         14         15
```

```
 16          17          18          19          20
# Matrix of size [4,5] [0x185a3d0 data= 0x17e18d0]
> output = b1:forward(input)
> = output
 2           4           6           8           10
 7           9           11          13          15
 12          14          16          18          20
 17          19          21          23          25
# Matrix of size [4,5] [0x185b370 data= 0x1718450]
> -- the bias component executes the following operation
> for i=1,input:dim(1) do input(i,':'):axpy(1.0, weights('b1'):transpose()) end
> = input
 2           4           6           8           10
 7           9           11          13          15
 12          14          16          18          20
 17          19          21          23          25
# Matrix of size [4,5] [0x185a3d0 data= 0x17e18d0]
```

9.6.1.3 dot_product

`ann.components.dot_product{ ... }`

The class `ann.components.dot_product` implements the dot product between a weights vector of every neuron and the given input vector, which is a vector-matrix product. If the input is a matrix with a bunch of patterns, the component executes a matrix-matrix product. The component contains a weights `matrix` with size `OxI`, where `O` is the number of neurons (output size), and `I` is the number of inputs (input size). The constructor receives:

- `name` is a string with the component name, **optional**.
- `weights` is a string with the weights name, **optional**.
- `input` is a number with the input size.
- `output` is the number of neurons.
- `transpose=false` is a boolean indicating if the weights `matrix` is transposed. It is **optional**, by default it is `transpose=false`.

```
> c = ann.components.dot_product{ weights='w1', input=4, output=5 }
> _,weights = c:build()
> weights('w1'):linspace()
> = weights('w1')
 1           2           3           4
 5           6           7           8
 9           10          11          12
 13          14          15          16
 17          18          19          20
# Matrix of size [5,4] [0x186e620 data= 0x182b050]
> input = matrix(3,4):linspace()
> = input
 1           2           3           4
 5           6           7           8
 9           10          11          12
# Matrix of size [3,4] [0x168f420 data= 0x1835190]
```

```
> output = c:forward(input)
> = output
 30          70          110         150         190
 70          174         278         382         486
 110         278         446         614         782
# Matrix of size [3,5] [0x185ee70 data= 0x18655c0]
> -- the performed operation is
> = input * weights('w1'):transpose()
 30          70          110         150         190
 70          174         278         382         486
 110         278         446         614         782
# Matrix of size [3,5] [0x1869f50 data= 0x1645e60]
```

In case of very sparse inputs, it is possible to replace the input `matrix` by a `tokens.sparse_matrix`, allowing to improve the efficiency of the operation. **Transformation of matrices into tokens and tokens into matrix is automatically performed**.

```
> -- a matrix with two rows:
> -- first row: active components are the 3 with 1, and the 2 with 0.5
> -- second row: active components are the 1 with 0.3
> dense_input = matrix(2,4):zeros():set(1,3,1):set(1,2,0.5):set(2,1,0.3)
> sparse_input = matrix.sparse( dense_input )
> = sparse_input
 0           0.5         1           0
 0.3         0           0           0
# SparseMatrix of size [2,4] in csr [0x17deaa0 data= 0x17864b0 0x17c9540 0x167cdb0], 3 non-zeros
> output = c:forward(input)
> = output
 4           10          16          22          28
 0.3         1.5         2.7         3.9         5.1
# Matrix of size [2,5] [0x18612d0 data= 0x17fb8a0]
> -- which is equivalent to the following
> output = c:forward(dense_input)
> = output
 4           10          16          22          28
 0.3         1.5         2.7         3.9         5.1
# Matrix of size [2,5] [0x185ee70 data= 0x1636e60]
```

9.6.1.4 hyperplane

`ann.components.hyperplane{ ... }`

The class `ann.components.hyperplane` is a wrapper around a `bias` and a `dot_product` components, implementing an hyperplane separator. The constructor receives:

- `name` an **optional** string with the component name.
- `dot_product` an **optional** string with the `dot_product` component name.
- `bias` an **optional** string with the `bias` component name.
- `dot_product_weights` an **optional** string with the `dot_product` component weights name.
- `bias_weights` an **optional** string with the `bias` component weights name.

- `input` a number with the input size.
- `output` a number with the input size.
- `transpose=false` a boolean indicating if the `dot_product` weights will be transposed in the operation.

```
> c = ann.components.hyperplane{ dot_product_weights='w1', bias_weights='b1',
                                  input=128, output=256 }
> _,weights = c:build()
> for name,w in pairs(weights) do print(name) print(w) end
w1
Large matrix, not printed to display
# Matrix of size [256,128] [0x185ee70 data= 0x16ae840]
b1
Large matrix, not printed to display
# Matrix of size [256,1] [0x1869120 data= 0x165d540]
```

9.6.2 Activation function components

9.6.2.1 logistic

```
ann.components.actf.logistic()
```

9.6.2.2 log_logistic

```
ann.components.actf.log_logistic()
```

9.6.2.3 softmax

```
ann.components.actf.softmax()
```

9.6.2.4 log_softmax

```
ann.components.actf.log_softmax()
```

9.6.2.5 tanh

```
ann.components.actf.tanh()
```

9.6.2.6 hardtanh

```
ann.components.actf.hardtanh()
```

9.6.2.7 relu

```
ann.components.actf.relu()
```

9.6.2.8 softplus

```
ann.components.actf.softplus()
```

9.6.2.9 sin

ann.components.actf.sin()

9.6.3 Container components

9.6.3.1 stack

ann.components.stack()

```
> ann.components.reset_id_counters() -- reset ID name generator
> mlp = ann.components.stack()
> mlp:push( ann.components.hyperplane{ input=100, output=200 } )
> mlp:push( ann.components.actf.logistic() )
> mlp:push( ann.components.hyperplane{ input=200, output=40 } )
> mlp:push( ann.components.actf.log_softmax() )
> _,weights = mlp:build()
> for name,w in pairs(weights) do print(name) print(w) end
w0
Large matrix, not printed to display
# Matrix of size [200,100] [0x1863df0 data= 0x1668030]
w2
Large matrix, not printed to display
# Matrix of size [40,200] [0x186bfd0 data= 0x17c71b0]
b1
Large matrix, not printed to display
# Matrix of size [200,1] [0x186aee0 data= 0x18159f0]
b3
Large matrix, not printed to display
# Matrix of size [40,1] [0x186d6d0 data= 0x175c910]
```

9.6.3.2 join

ann.components.join()

9.6.4 Filter components

9.6.4.1 dropout

ann.components.dropout()

```
> c = ann.components.dropout{ random=random(3284), prob=0.5, value=0.0 }
```

9.6.4.2 select

ann.components.select()

9.6.4.3 slice

ann.components.slice()

9.6.4.4 gaussian_noise

```
ann.components.gaussian_noise{ random, prob, var, mean }
```

9.6.4.5 salt_and_pepper

```
ann.components.salt_and_pepper{ random, prob, zero, one }
```

9.6.5 Convolutional components

This components are used to build Convolutional Neural Networks. If you use `dataset.matrix`, your patterns will be flattened at converted into a one dimensional matrix. This forces to add a `rewrap` components at the beginning of your ANN. Follows an example of a FULL CNN for MNIST task (28x28 pixels, images of digits):

```
-- tables for the CNN configuration
ishape  = {1, 28, 28} -- for input matrix rewrapping
conv1   = {1, 5, 5} nconv1=20
maxp1   = {1, 2, 2}
conv2   = {nconv1, 5, 5,} nconv2=50
maxp2   = {1, 2, 2}
hidden  = 500

thenet = ann.components.stack():
push( ann.components.rewrap{ size=ishape } ):
push( ann.components.convolution{ kernel=conv1, n=nconv1 } ):
push( ann.components.convolution_bias{ n=nconv1, ndims=#conv1 } ):
push( ann.components.actf.tanh() ):
push( ann.components.max_pooling{ kernel=maxp1,} ):
push( ann.components.convolution{ kernel=conv2, n=nconv2 } ):
push( ann.components.convolution_bias{ n=nconv2, ndims=#conv2 } ):
push( ann.components.actf.tanh() ):
push( ann.components.max_pooling{ kernel=maxp2 } ):
push( ann.components.flatten() )

-- using the method precompute_output_size, it is possible to know
-- the size of the convolution after the flatten operation
local conv_size = thenet:precompute_output_size()[1]

thenet:
push( ann.components.hyperplane{ input=conv_size, output=hidden } ):
push( ann.components.actf.tanh() ):
push( ann.components.hyperplane{ input=hidden, output= 10 } ):
push( ann.components.actf.log_softmax() )
```

9.6.5.1 convolution

```
ann.components.convolution{ kernel, step, n, name, weights, ...  }
```

A convolutional component could be created as:

```
> c = ann.components.convolution{ kernel={3, 5, 5}, step={1, 1, 1}, n=10,
                                  name="conv-W1", weights="W1",
                                  input_planes_dim=1 }
```

This component executes a convolution using the given `kernel` sizes, moving the convolution window following `step` table, and using `n` different kernels. This module has a dynamic input/output size, the convolution is performed over all the input following the indicated parameters.

- `input_planes_dim` is a number (**optional**, by default is 1) which indicates the dimension **K** at input matrix where are located the input planes.
- `kernel` is a table which describes the size of each kernel. The **K** element of this table is always the number of PLANES at the input matrix. Therefore, a kernel over a 1-dim signal will be like `kernel={1, 5}` being **K=1**. For a 2D image will be `kernel={1, 5, 5}`, for a 2D image with RGB color will be `kernel={3, 5, 5}` if **K=1**, otherwise it could be `kernel={5, 3, 5}` if **K=2** or `kernel={5, 5, 3}` if **K=3**. For a RGB video sequence the kernel will be `kernel={3, 5, 5, 5}` for **K=1**, and so on.
- `step` is a table which indicates how to move the kernel. The number of steps at each dimension will be `(input_dim[i] - kernel[i])/step[i] + 1`. The **K** element of this table is forced to be `1`, so that is the number of planes at input matrix. The `step` is **optional**, by default has all its elements assigned to `1`.
- `n` is the number of kernels to be applied. It is the number of output planes produced by this component (number of neurons).
- `name` and `weights` are the strings with for search components and connection objects.

The output produced by this component will be of:

- output_size[1]=`n`
- output_size[i+1]=(input_size[i] - kernel[i])/step[i] + 1, FOR i=1,...,`input_planes_dim`-1
- output_size[i]=(input_size[i] - kernel[i])/step[i] + 1, FOR i=`input_planes_dim`+1,...,`#kernel`

By default, `input_planes_dim=1`, so the output size will be simplified as:

- output_size[1]=`n`
- output_size[i]=(input_size[i] - kernel[i])/step[i] + 1, FOR i=2,...,`#kernel`

9.6.5.2 convolution_bias

```
ann.components.convolution_bias{ n, ndims, name, weights }
```

```
> c = ann.components.convolution_bias{ n=10, ndims=3,
                                       name="conv-B1", weights="B1" }
```

- `n` is the number of planes at the input (the first dimension size of the input matrix).
- `ndims` is the number of dimensions expected at the input matrix.
- `name` and `weights` as usual

9.6.5.3 max_pooling

```
ann.components.max_pooling{ kernel, name }
```

```
> c = ann.components.max_pooling{ kernel={1, 2, 2}, name="pool-2" }
```

- `kernel` is a table with the sizes of the kernel applied to the input matrix. Depending on this the behavior of the max-pooling could be to do a down-sampling of an input matrix (as in the example), or to convert the input in a fixed size feature vector (`kernel = {1, 0, 0}`). The `0` value at one component means to fit this dimension with the same dimension of input matrix. So, the last example `{1, 0, 0}` will be a max-pooling computed over all positions for each input plane, producing as output a feature vector of INPUT PLANES size.
- `name` as usual.

9.6.5.4 flatten

```
ann.components.flatten{ [name] }
```

This components converts an input matrix formed by N patterns of any dimensionality to an output bidimensional matrix with N rows and M columns, where M is the product of all input matrix dimensions (except the first one which is the number of patterns).

```
> c = ann.components.flatten{ name="flatten" }
```

9.6.6 Other components

9.6.6.1 copy

```
ann.components.copy
```

Chapter 10

ann.loss package

Related with module `require("aprilann.ann.loss")`.

This package defines the loss functions included in APRIL-ANN. All loss functions share the same interface. Normally, they are implemented in C++ and binded to Lua.

The interface of loss functions is the following:

- `loss,loss_matrix = loss:compute_loss(input,target)`: this method computes the loss between two `tokens`, the `input` and the `target`. Normally they are bi-dimensional `matrix` instances with size NxM, where N is the number of patterns in the bunch (mini-batch), and M is the number of outputs in the ANN component. The method returns two values, the `loss`, which is a number with the mean loss in the given bunch of patterns. The `loss_matrix`, which is a one-dimensional `matrix` of size N containing the loss for every pattern. In some cases, as for example in FMeasure-based loss functions, this loss `matrix` is of size 1, because the loss function is computed over the bunch of patterns, and is not separable.
- `gradient = loss:gradient(input,target)`: this method computes the gradient of the loss between the two `input` and the `target`. It returns a bi-dimensional matrix with size NxM. Each component of this `matrix` is the partial derivative ANN outputs respect to the loss function.
- `loss,loss_matrix = loss:accum_loss(loss,loss_matrix)`: this method receives the output of `compute_loss` method, and accumulates the given loss in its internal state. It is useful to compute the loss of a large number of patterns.
- `loss_matrix = loss:accum_loss(loss_matrix)`: this method is a specialization of the previous one, but receiving only the `loss_matrix`.
- `mean,variance = loss:get_accum_loss()`: this method returns two numbers, the `mean` and the `variance` of the accumulated loss in the internal state of the loss function object.

Tt is possible to develop new loss functions by implementing Lua classes derived from `ann.loss` class, following this example:

```
> myloss,myloss_methods = class("myloss",ann.loss)
> function myloss:constructor()
    -- Your code to initialize self reference
  end
> function myloss_methods:compute_loss(input,target)
    -- YOUR CODE
```

```
    return loss,loss_matrix
  end
> function myloss_methods:gradient(input,target)
    -- YOUR CODE
    return gradient_token
  end
> function myloss_methods:accum_loss(loss,loss_matrix)
    local loss_matrix = loss_matrix or loss
    -- YOUR CODE
    return loss or loss_matrix, loss_matrix
  end
> function myloss_methods:get_accum_loss()
    -- YOUR CODE
    return loss_mean,loss_variance
  end
> function myloss_methods:reset()
    -- YOUR CODE
  end
> function myloss_methods:clone()
    -- YOUR CODE
    return cloned_obj
  end
```

10.1 Mean squared error (MSE)

This loss function is defined at the object `ann.loss.mse`:

```
> loss = ann.loss.mse()
```

The constructor could receive an **optional** parameter with the expected number of outputs at the ANN component. *If given*, it will be used as sanity check forcing to be equal to the given input/target sizes. *If not given*, the size check will be ignored.

This loss function computes the mean squared error between the given input/target patterns. It computes the following expression:

$$J = \frac{1}{2N} \sum_{i,j} \left(h_i^j - t_i^j \right)^2$$

Figure 10.1: MSE

Where `N` is the number of patterns, `h_i^j` is the position `(i,j)` in the input matrix (pattern `i`, component `j`), and `t_i^j` is the same position at the target matrix.

10.2 Mean absolute error (MAE)

This loss function is defined at the object `ann.loss.mae`:

```
> loss = ann.loss.mae()
```

The constructor could receive an **optional** parameter with the expected number of outputs at the ANN component. *If given*, it will be used as sanity check forcing to be equal to the given input/target sizes. *If not given*, the size check will be ignored.

This loss function computes the mean absolute error between the given input/target patterns. It computes the following expression:

$$J = \frac{1}{NM} \sum_{i,j} |h_i^j - t_i^j|$$

Figure 10.2: MAE

Where `N` is the number of patterns, `M` is the number of outputs, `h_i^j` is the position `(i,j)` in the input matrix (pattern `i`, component `j`), and `t_i^j` is the same position at the target matrix.

10.3 Cross entropy

This loss function is defined at the object `ann.loss.cross_entropy`:

```
> loss = ann.loss.cross_entropy()
```

The constructor could receive an **optional** parameter with the expected number of outputs at the ANN component. *If given*, it will be used as sanity check forcing to be equal to the given input/target sizes. *If not given*, the size check will be ignored.

This object is implemented to work **only** with `log_logistic` activation function. This loss function computes the cross entropy between the given input/target patterns, interpreting the ANN component output as a **binomial** distribution. It computes the following expression:

$$J = \frac{1}{N} \sum_{i,j} - \left[t_{ij} \log h_i^j + (1 - t_{ij}) \log(1 - h_i^j) \right]$$

Figure 10.3: CE

Where `N` is the number of patterns, `h_i^j` is the position `(i,j)` in the input matrix (pattern `i`, component `j`, in natural scale), and `t_i^j` is the same position at the target matrix.

10.4 Multi-class cross entropy

This loss function is defined at the object `ann.loss.multi_class_cross_entropy`:

```
> loss = ann.loss.multi_class_cross_entropy()
```

The constructor could receive an **optional** parameter with the expected number of outputs at the ANN component. *If given*, it will be used as sanity check forcing to be equal to the given input/target sizes. *If not given*, the size check will be ignored.

$$J = \frac{1}{N} \sum_{i,j} -t_{ij} \log h_i^j$$

Figure 10.4: CE

This object is implemented to work **only** with `log_softmax` activation function. This loss function computes the cross entropy between the given input/target patterns, interpreting the ANN component output as a **multinomial** distribution. It computes the following expression:

Where `N` is the number of patterns, `h_i^j` is the position `(i,j)` in the input matrix (pattern `i`, component `j`, in natural scale), and `t_i^j` is the same position at the target matrix.

10.5 Macro averaging multi-class F-Measure

This loss function is defined at the object `ann.loss.batch_fmeasure_macro_avg`:

```
> loss = ann.loss.batch_fmeasure_macro_avg{ beta=0.5 }
```

The constructor could receive an **optional** table parameter with the following fields:

- `size=0`: expected number of outputs at the ANN component. *If given*, it will be used as sanity check forcing to be equal to the given input/target sizes. *If not given*, the size check will be ignored.
- `beta=1`: the `b` parameter in the F-Measure expression below. By default it is set to 1.
- `complement=false`: a boolean indicating if the input/target values must be computed complemented (1 - value), swapping positive and negative classes.

This object is implemented to work with `logistic` or `softmax` activation function. This loss function computes the F-Measure between the given input/target patterns, interpreting the ANN component output as a **multinomial** distribution. It computes the following expression:

$$J = \frac{1}{M} \sum_j \frac{(1+b^2) h_j \cdot t_j}{(\sum o_j) + b^2 (\sum t_j)}$$

Figure 10.5: FM

Where `M` is the number of outputs, `b` is the beta parameter of the F-mesaure, `h_j · t_j` is the dot product between column vectors with input/target values of class `j`, and `sum(o_j)` and `sum(t_j)` is the sum of all the elements in the column vectors.

10.6 Micro averaging multi-class F-Measure

This loss function is defined at the object `ann.loss.batch_fmeasure_micro_avg`:

```
> loss = ann.loss.batch_fmeasure_micro_avg{ beta=0.5 }
```

The constructor could receive an **optional** table parameter with the following fields:

- `size=0`: expected number of outputs at the ANN component. *If given*, it will be used as sanity check forcing to be equal to the given input/target sizes. *If not given*, the size check will be ignored.
- `beta=1`: the `b` parameter in the F-Measure expression below. By default it is set to 1.
- `complement=false`: a boolean indicating if the input/target values must be computed complemented (1 - value), swapping positive and negative classes.

This object is implemented to work with `logistic` activation function. This loss function computes the F-Measure between the given input/target patterns, interpreting the ANN component output as a **binomial** distribution. If it is used with `softmax` (**multinomial** distribution), then this function computes accuracy. It follows this expression:

$$J = \frac{(1 + b^2)h \cdot t}{(\sum o) + b^2(\sum t)}$$

Figure 10.6: FM

Where `b` is the beta parameter of the F-mesaure, `dot(h,t)` is the dot product between the input/target matrices re-interpreted as two column vectors, and `sum(o)` and `sum(t)` is the sum of all the elements in the matrices.

10.7 Zero-one loss function

This loss function is defined at the object `ann.loss.zero_one`:

```
> loss = ann.loss.zero_one([nil, [, 0.5 ] ])
```

The constructor could receive an **optional** first parameter with the expected number of outputs at the ANN component. *If given*, it will be used as sanity check forcing to be equal to the given input/target sizes. *If not given*, the size check will be ignored.

It could receive an **optional** second parameter which by default is `0.5`. This second parameter is the threshold which defines when the output is taken as 1. **NOTE** that if you are using `log_logistic` outputs, this threshold must be set to `math.log(0.5)`. This parameter is only useful when the model has one output, that is, for **two-class** problems.

This object is **not derivable**, so the `compute_gradient` method is forbidden. The loss function could be use to compute validation error, but not for training. It computes the accuracy of the model classifying to the class with maximum probability.

Chapter 11

ann.optimizer package

11.1 Introduction

Related with the module `require("aprilann.ann.optimizer")`.

The optimizer is an object which implements the learning algorithm. Every class in `ann.optimizer` is an optimizer. Several learning hyperparameters are available, depending in the selected optimizer. This learning hyperparameters are known as *options*, and could be set **globally** (to all the connection weight layers of the ANN), or **layerwise** (to a concrete connection weights object, identified by its name). Optimizers implement the following API.

11.1.1 Interface of ann.optimizer classes

11.1.1.1 clone

```
other = opt:clone()
```

Returns a deep copy of the caller object.

11.1.1.2 get_option

```
value = opt:get_option(option)
```

Return the *global* value of a given learning `option` name.

11.1.1.3 set_option

```
opt:set_option(option, value)
```

Sets the *global* value of a given learning `option` name.

11.1.1.4 set_layerwise_option

```
opt:set_layerwise_option(layer, option, value)
```

Sets a *layerwise* `option` for the given `layer` name.

11.1.1.5 get_layerwise_option

```
value = opt:get_layerwise_option(layer, option)
```

Returns the *layerwise* `option` of the given `layer` name.

11.1.1.6 get_option_of

```
value = opt:get_option_of(layer, option)
```

Returns the option which is applicable to the given `layer` name. If a *layerwise* option was previously defined, the method returns its value. Otherwise, the value of the *global* option will be returned.

11.1.1.7 execute

```
loss,gradients,...  = opt:execute(eval,cnn)
```

This is the core function of optimizer objects. It receives two parameters, and return the same values as returned by `eval` function. The parameters are:

- `eval` is a Lua function which returns at least the first two parameters of the list below. The function receives two inputs:
  1. A table of candidate weigths to compute the loss.
  2. The iteration number of the algorithm.

This function could return more values, all of them will be returned by the call to execute:

```
1. The loss of a bunch of data (a mini-batch).

2. A gradients matrix dictionary (as returned by ANN components method
`compute_gradients()`). It must be a table of names=>`matrix`.
```

- `cnn` a dictionary of matrices with the connection weight objects, as the dictionary returned by ANN components method `copy_weights()`. It must be a table of names=>`matrix`. The `execute` method can modify the matrices contained at `cnn`, by following the given gradients, can use a bunch of weight candidates and call `eval` function to test the loss.

11.2 Coupling ANNs with optimizer

11.2.1 Using a trainer object

The `trainable.supervised_trainer` class is the default coupling between the optimizer, the ANN component and the loss function. The optimizer could be set at the constructor:

```
> trainer = trainable.supervised_trainer(ann_component, ann.loss.mse(),
                                         bunch_size, ann.optimizer.sgd())
```

The hyperparemters of `optimizer` objects can be modified by the `trainer` object:

- `trainer:set_option(option,value)`: sets a global learning option value.

- `value=trainer:get_option(option)`: gets a global learning option value.
- `trainer:set_layerwise_option(layer_match,option,value)`: sets a layerwise learning option value of all the connection weight objects whose name *matches* the given `layer_match` Lua pattern string.
- `value=trainer:get_option_of(layer,option)`: gets the option value applicable to the given layer.

```
trainer = trainable.supervised_trainer(ann_component, ann.loss.mse(),
                                       bunch_size, ann.optimizer.sgd())
trainer:build()
trainer:set_option("learning_rate", number)
trainer:set_option("momentum", number)
-- regularization is recommended to not be applied at bias connections
trainer:set_layerwise_option("w.*", "weight_decay", number)
trainer:set_layerwise_option("w.*", "max_norm_penalty", number)
trainer:set_layerwise_option("w.*", "L1_norm", number)
```

Doing this, when you call `trainer:train_dataset(...)` or `trainer:validate_dataset(...)`, the object implements a default `eval` function for the optimizer.

11.2.2 From the *scratch*

First, you need to construct the ANN component, the loss function and the optimizer instance, and the configuration of the optimizer.

```
local thenet = ann.mlp.all_all.generate("256 inputs 128 tanh 10 log_softmax")
local thenet,cnns = thenet:build()
local loss = ann.loss.multi_class_cross_entropy()
local opt  = ann.optimizer.sgd()
opt:set_option("learning_rate", 0.01)
```

The initialization of the weights is required:

```
local rnd = random(1234)
for _,w in pairs(cnns) do w:uniformf(-0.1,0.1,rnd) end
```

The training is performed over random data, generated on-the-fly:

```
local M = matrix.col_major -- ANNs need col_major matrices
local weight_grads = {} -- upvalue for eval function
for i=1,1000 do
  local input  = M(1,256):uniformf(0,1,rnd)
  local target = M(1,10):zeros():set(1, rnd:randInt(1,10), 1.0)
  opt:execute(function(weights,it)
                    if cnns ~= weights then thenet:build(weights) cnns=weights end
                    thenet:reset(it)
                    local out = thenet:forward(input)
                    local tr_loss,tr_matrix = loss:compute_loss(out,target)
                    thenet:backprop(loss:gradient(out,target))
                    weight_grads:zeros()
                    weight_grads = thenet:compute_gradients(weight_grads)
                    return tr_loss,weight_grads
                end,
                cnns)
end
```

All this code could be modified in many ways, to customize or implement your own methods.

NOTE the full code of this example is available at: EXAMPLES/optimizer-from-scratch.lua

11.3 Stochastic Gradient Descent

The `ann.optimizer.sgd` class trains the neural network following the *Stochastic Grandient Descent* algorithm. It incorporates regularization and momentum hyperparameters.

```
> opt = ann.optimizer.sgd()
> loss,gradients = opt:execute(function(weights,it)
                                 -- HERE YOUR CODE
                                 return loss,gradients
                               end,
                               -- The connections
                               cnns)
```

Its options are (default values are indicated with `=` symbol):

- `learning_rate`: the learning rate controls the portion of the gradient used to update the weights.
- `decay=1e-05`: controls the decay of learning rate.
- `momentum=0`: is a inertial hyperparameter which applies a portion of the weight update in the previous iteration.
- `weight_decay=0`: a L2 regularization term.
- `L1_norm=0`: a L1 regularization term, a naive implementation with ZERO truncation to avoid ZERO cross.
- `max_norm_penalty=0`: a constrain penalty based on the two-norm of the weights.

The algorithm uses the following learning rule:

```
w = w - lr'*( grad(L)/grad(w') + weight_decay*w' + L1_norm*sign(w') ) + momentum*(w' -
w")
```

where `w`, `w'` and `w"` are the weight values at next, current, and previous iterations; `lr'` is the `learning_rate`, and `grad(L)/grad(w')` is the gradient of the loss function at the given weight. The L1 regularization is performed following truncate gradient algorithm. After this learning rule, the constraint `max_norm_penalty` is applied, forcing the 2-norm of the input weights of every neuron to be less than the given parameter.

```
> opt = ann.optimizer.sgd() -- an instance
```

11.4 Averaged Stochastic Gradient Descent

The `ann.optimizer.asgd` class trains the neural network following the *Averaged Stochastic Grandient Descent* algorithm, taken from Leon Bottou, *Stochastic Gradient Descent Tricks*, Microsoft Research, 2012. It incorporates regularization and learning rate decay hyperparameters.

```
> opt = ann.optimizer.asgd()
> loss,gradients = opt:execute(function(weights,it)
                                 -- HERE YOUR CODE
                                 return loss,gradients
                               end,
                               -- The connections
                               cnns)
```

Its options are (default values are indicated with = symbol):

- `learning_rate`: the learning rate controls the portion of the gradient used to update the weights.
- `lr_decay=0.75`: controls the learning rate decay. The effective learning rate is computed as `lr / (1 + lr*t)^lr_decay`, being `t` the number of presentations, which has not to be confused with number of epochs. Number of presentations would be higher of number of epochs.
- `t0=0`: when to start averaging. Before `t0`, the algorithm will be *SGD*. As before, `t0` is indicated in number of presentations, i.e. one epoch will be `t0 = ceil( number_of_patterns / bunch_size )`. It is recommended to set this option to at least one epoch.
- `weight_decay=0`: a L2 regularization term.

```
> opt = ann.optimizer.asgd() -- an instance
```

11.5 Resilient backpropagation (RProp)

The `ann.optimizer.rprop` class trains the neural network following the *RProp* algorithm. It incorporates an option to set the number of iterations done with every mini-batch of patterns. This optimizer has a step size hyperparameter for each weight parameter in the model. All options are global, layer-wise options doesn't make sense here.

```
> opt = ann.optimizer.rprop()
> loss,gradients = opt:execute(function(weights,it)
                                 -- HERE YOUR CODE
                                 return loss,gradients
                               end,
                               -- The connections
                               cnns)
```

Its options are (default values are indicated with = symbol):

- `initial_step=0.1`: the initial step size of every weight.
- `eta_plus=1.2`: value which controls how much proportion is the step increased when the gradient sign is the same between two iterations.
- `eta_minus=0.5`: value which controls how much proportion is the step decreased when the gradient sign changes between two iterations.
- `max_step=50`: the maximum value for the weight step hyperparameter.
- `min_step=1e-05`: the minimum value for the weight step hyperparameter.
- `niter=1`: the number of iterations done with one mini-batch of patterns.

The algorithm modifies the `step(i)` parameter of a weight at iteration `i`:

```
          | step(i-1) * eta_minus, iff sign(grad(w')/grad(L)) <> sign(grad(w'')/grad(L))
step(i) = | step(i-1) * eta_plus,  iff sign(grad(w')/grad(L)) == sign(grad(w'')/grad(L))
```

It updates the weight following this equation:

```
w = w' - sign(grad(w')/grad(L)) * step(i)
```

In the above equations, `w`, `w'` and `w"` are the weight values at next, current, and previous iterations and `sign(grad(.)/grad(L))` is the sign of the gradient of the loss function at the given weight. Note that the `step` is saturated with the values `max_step` and `min_step`.

```
> opt = ann.optimizer.rprop() -- an instance
```

11.6 Conjugate Gradient (CG)

The `ann.optimizer.cg` class trains the neural network following the *CG* algorithm. It is a second order Hessian Free optimizer. Its convergence is usually faster than SGD algorithm.

```
> opt = ann.optimizer.cg()
> loss,gradients = opt:execute(function(weights,it)
                                 -- HERE YOUR CODE
                                 return loss,gradients
                               end,
                               -- The connections
                               cnns)
```

Its options are (default values are indicated with `=` symbol):

- `rho=0.01`: Constant for Wolf-Powell conditions (*global option*).
- `sig=0.5`: Constant for Wolf-Powell conditions (*global option*).
- `int=0.1`: Reevaluation limit (*global option*).
- `ext=3`: Maximum number of extrapolations (*global option*).
- `max_iter`: Maximum number of iterations (*global option*).
- `max_eval=1.25*max_iter`: Maximum number of evaluations (*global option*).
- `ratio=100`: Maximum slope ratio (*global option*).
- `weight_decay`: Weights L2 regularization (*global and layer-wise option*).
- `L1_norm`: Weight L1 regularization (*global and layer-wise option*).
- `max_norm_penalty`: Weight max norm upper bound (*global and layer-wise option*).

This implementation is rewrite of Torch 7 optim package, which is a rewrite of `minimize.m` written by Carl E. Rasmussen.

```
> opt = ann.optimizer.cg() -- an instance
```

11.7 Quickprop

The `ann.optimizer.quickprop` class trains the neural network following the *Quickprop* algorithm. It is a second order optimizer which uses a quadratic approximation to speed-up the learning convergence. It is usually faster than SGD, but can suffer of chaotic oscillations.

```
> opt = ann.optimizer.quickprop()
> loss,gradients = opt:execute(function(weights,it)
                                 -- HERE YOUR CODE
                                 return loss,gradients
                               end,
                               -- The connections
                               cnns)
```

Its options are (default values are indicated with `=` symbol):

- `learning_rate`: It is mandatory to be given.
- `mu=1.75`: Maximum growth factor.
- `epsilon=1e-04`: Bootstrap factor.
- `max_step=1000`: Maximum step value.
- `weight_decay`: Weights L2 regularization.
- `L1_norm`: Weight L1 regularization.
- `max_norm_penalty`: Weight max norm upper bound.

```
> opt = ann.optimizer.quickprop() -- an instance
```

11.8 AdaDelta

The `ann.optimizer.adadelta` class trains the neural network following the *AdaDelta* algorithm. It is a method which dynamically adapts the learning rate just using first order information. It as simple as SGD, and appears to be more robust.

```
> opt = ann.optimizer.adadelta()
> loss,gradients = opt:execute(function(weights,it)
                                 -- HERE YOUR CODE
                                 return loss,gradients
                               end,
                               -- The connections
                               cnns)
```

Its options are (default values are indicated with `=` symbol):

- `decay=0.95`: Decay of the accumulated gradient and updates.
- `epsilon=1e-06`: To avoid numerical issues.
- `weight_decay=0.0`: Weights L2 regularization.
- `max_norm_penalty=0.0`: Weight max norm upper bound.

```
> opt = ann.optimizer.adadelta() -- an instance
```

Chapter 12

ann.graph package

12.1 Introduction

This package contains an implementation of ANNs by using a graph description. The graph allow to declare *delayed* connections, which can be used to develop recurrent neural networks as LSTMs or Elman.

12.2 Graph based ANNs

Graph ANNs are declared as instances of the class `ann.graph`. The ANN graph is a model where nodes are ANN componentes (any object of `ann.components` and objects defined in `ann.graph` and `ann.graph.blocks`). So, nodes have an input and an output tokens, in the same way as ANN componentes receive a token and produce as output another token. Nodes can receive any number of connections as input and its output can be connected to other multiple nodes. When multiple input connections are received, they are put together into a `tokens.vector.bunch` instance. The graph implements properly the propagation of gradients between the nodes, and uses the methods `forward`, `backprop` and `compute_gradients` of the components in every node.

The graph object is considered itself as an ANN component, allowing to declare graphs where nodes are other graphs. Every graph has two special nodes, 'input' and 'output', which are used to connect the visible parts of the ANN.

In ANN graphs loops are allowed, but they cannot be made of normal connections, and the concept of *delayed* connections is introduced (indeed, normal connections are whose with `delay=0`). An ANN graph with delayed connections is equivalent to the concept of Recurrent Neural Network (RNN in the following).

The training of RNNs is done following the Back-Propagation Trough Time (BPTT) algorithm. RNNs have an special behavior in `forward` method, the graph takes note of the state (input, output, gradient deltas, ...) for every node, allowing to take as input the activation in any past instant. The `backprop` method returns a `tokens.null` instance, that is, its output is none, this method just annotates the given input error deltas for a future use. Calling the method `compute_gradients` the error deltas given at `backprop` are propagated through all the space and time, and the weight gradients are computed.

12.2.1 Constructor

```
g = ann.graph( [ name ] )
```

The constructor an optional `name` argument.

12.2.2 connect

`g:connect( source, dest1, ..., [ delay=0 ] )`

This method connects a path of nodes in the graph. It receives as arguments:

- `source`: The first node in the path (`source`). It can be an ANN component or the string `'input'`.
- `dest1`: The second node in the path (`dest1`). It can be an ANN component or the string `'output'`.
- `...`: A variadic list of arguments with zero or more nodes which form the path. Every node in this list can be an ANN component or `output`.
- `delay=0`: The last argument is optional and by default it is zero. This argument is needed to declare *delayed* connections. All the connections in the path would be declared with the delay given in this argument. **Note** that normally only the connection between two nodes need to be delayed, and for this purpose the method `g:delayed(source,destination)` has been declared.

For correction, a graph is valid only if the `input` node is a source and `output` node is a sink, and every node is reachable from the `input`.

The following example shows how to declare a Jordan network.

```
> g        = ann.graph()
> bind     = ann.graph.bind()
> out_actf = ann.components.logistic()
> g:connect('input', bind,
            ann.components.hyperplane{ input=2, output=4 },
            ann.components.actf.logistic(),
            ann.components.hyperplane{ input=4, output=1 },
            out_actf, 'output')
> g:delayed(out_actf, bind) -- recurrent connection
> g:build()
```

12.2.3 delayed

`g:delayed( source, destination )`

This is equivalent to `g:connect(source, destination, 1)`.

12.2.4 show_nodes

`g:show_nodes()`

This method is used for *debug* purposes, and show all the nodes in the given graph. If the graph contains other graphs as nodes, their nodes would be shown recursively. The output indicates the level of recursion, the name of the component in the corresponding node, and the type of the object, as in the following example (extracted from a LSTM test):

```
# (level) name type
(0) a2                          ann.components.actf.log_logistic
(0) parity::output              string
(0) LSTM                        ann.graph +
  (1) LSTM::f::layer              ann.components.hyperplane
  (1) LSTM::input                 string
  (1) LSTM::o::actf               ann.components.actf.logistic
```

```
  (1) LSTM::i::peephole             ann.graph.bind
  (1) LSTM::o::gate                 ann.graph.cmul
  (1) LSTM::o::layer                ann.components.hyperplane
  (1) LSTM::f::gate                 ann.graph.cmul
  (1) LSTM::actf                    ann.components.actf.softsign
  (1) LSTM::memory                  ann.graph.add
  (1) LSTM::o::peephole             ann.graph.bind
  (1) LSTM::i::actf                 ann.components.actf.logistic
  (1) LSTM::cell_input              ann.components.hyperplane
  (1) LSTM::output                  string
  (1) LSTM::i::gate                 ann.graph.cmul
  (1) LSTM::f::actf                 ann.components.actf.logistic
  (1) LSTM::i::layer                ann.components.hyperplane
  (1) LSTM::f::peephole             ann.graph.bind
(0) l2                            ann.components.hyperplane
(0) parity::input                 string
```

12.2.5 dot_graph

`g:dot_graph(filename)`

This method is for *debug*. It writes to the given `filename` a DOT graph which can be transformed in PDF using graphviz.

12.2.6 build

```
g,weights,components = g:build( [ table ] )
```

See ANN package doc.

12.2.7 forward

```
output = g:forward( input, [ during_training=false ] )
```

See ANN package doc.

12.2.8 backprop

```
output = g:backprop( input )
```

See ANN package doc.

Note that this method changes its default behavior when the graph is a RNN. In this case, the output of this method is a `tokens.null` instance, so it can be ignored.

12.2.9 compute_gradients

```
table = g:compute_gradients( [table] )
```

See ANN package doc.

12.2.10 bptt_backprop

```
table = g:bptt_backprop()
```

Forces the BPTT algorithm execution, and returns a table (Lua array) with the delta gradients at the ANN input for every time instant.

12.2.11 get_bptt_state

```
table = g:get_bptt_state()
```

Returns a table with the state of the whole ANN for every time instant.

```
table = g:get_bptt_state(time)
```

Returns a table with the state of the whole ANN for the given time instant.

12.2.12 reset

```
g:reset( [ n ] )
```

See ANN package doc.

Additionally with the standard behavior, this method reinitializes the BPTT tables, and must be called before starting a new sequence.

12.2.13 get_is_recurrent

```
boolean = g:get_is_recurrent()
```

Indicates if the caller graph is recurrent or not.

12.2.14 set_bptt_truncation

```
g:set_bptt_truncation( backstep )
```

Changes the BPTT algorithm behavior, truncating the gradient computation every `backstep` number of iterations. This value can be `math.huge` to indicate an infinite limit. Besides this value, another usual one is `1`, which transforms allow to use the algorithm as a kind of on-line learning algorithm.

12.3 Graph junctions

12.3.1 bind

```
ann.graph.bind{ [ name=string ], [ input=number ], [ output=number ], [ size=number ] }
```

12.3.2 add

```
ann.graph.add{ [ name=string ], [ input=number ], [ output=number ] }
```

12.3.3 cmul

```
ann.graph.cmul{ [ name=string ], [ input=number ], [ output=number ] }
```

12.3.4 index

```
ann.graph.index(n, { [ name=string ], [ input=number ], [ output=number ] })
```

12.4 Graph blocks constructors

12.4.1 Elman

```
ann.graph.blocks.elman{ [ name=string ], [ input=number ],
                        [ output=number ], [ actf=string ] })
```

12.4.2 LSTM

```
ann.graph.blocks.lstm{ [ name=string ], [ input=number ],
                       [ output=number ], [ actf=string ],
                       [ peepholes=true ], [ input_gate=true ],
                       [ forget_gate=true], [ output_gate=true ] })
```

Chapter 13

ann.autoencoders package

13.1 Introduction

Package `autoencoders` could be loaded via the standalone binary, or in Lua with `require("aprilann.autoencoders)`.

Stacked Denoising Auto-Encoders (SDAE) are a kind of deep neural network which is pre-trained following greedy layerwise algorithm but introducing at noise input of each layerwise auto-encoder. Some function facilities are implemented to help with the training of SDAE.

13.2 Greedy layerwise pre-training of SDAE

Greedy layerwise pre-training consists in train each pair of layers, from input to output, in a greedy way (see Paper SDAE, 2010, Vincent Pascal et al.). Pre-training receives as input a table with parameters of training algorithm. For example, a table like this:

```
layers = {
  { size= 256, actf="logistic"}, -- INPUT
  { size= 256, actf="logistic"}, -- FIRST HIDDEN LAYER
  { size= 128, actf="logistic"}, -- SECOND HIDDEN LAYER
  { size=  32, actf="logistic"}, -- THIRD HIDDEN LAYER
}
perturbation_random = random(824283)
params_pretrain = {
  input_dataset         = train_input,  -- a dataset which is the input of the autoencoders
  replacement           = nil,          -- a number (or nil) indicating replacement
  on_the_fly            = false,        -- a boolean (or nil) for on-the-fly
  shuffle_random        = random(1234), -- for shuffle durint backpropagation
  weights_random        = random(7890), -- for weights random initialization
  layers                = layers,       -- layers description
  supervised_layer      = { size = 10, actf = "log_softmax" }, -- it is possible to pre-train supervise
  output_datasets       = { train_output }, -- the output dataset
  bunch_size            = bunch_size,       -- the size of the mini-batch
  optimizer             = function() return ann.optimizer.sgd() end, -- optimizer function
  training_options      = { -- this table contains learning options and dataset noise filters
    -- global options
    global = {
      -- pure ANN learning hyperparameters
```

```
      ann_options = { learning_rate = 0.01,
                      momentum      = 0.02,
                      weight_decay  = 1e-05 },
      -- noise filters (a pipeline of filters applied to input in order). Each one must be a dataset
      noise_pipeline = { function(ds) return dataset.perturbation{ -- gaussian noise
                              dataset  = ds,
                              mean     = 0,    -- gaussian mean
                              variance = 0.01, -- gaussian variance
                              random   = perturbation_random } end,
                         function(ds) return dataset.salt_noise{ -- salt noise (or mask noise)
                              dataset  = ds,
                              vd       = 0.10, -- percentage of values masked
                              zero     = 0.0,  -- mask value
                              random   = perturbation_random } end },
      min_epochs            = 4,
      max_epochs            = 200,
      pretraining_percentage_stopping_criterion = 0.01,
    },
    -- it is possible to overwrite global values with layerwise dependent values (also noise_pipeline)
    layerwise = { { min_epochs=50 },  -- first autoencoder pretraining
                  { min_epochs=20 },  -- second autoencoder pretraining
                  { ann_options = { learning_rate = 0.04,
                                    momentum      = 0.02,
                                    weight_decay  = 4e-05 },
                    min_epochs=20 },    -- third autoencoder pretraining
                  { min_epochs=10 }, }, -- supervised pretraining
  }
}
```

Fields `supervised_layer` and `output_datasets` are optional. If they are given, the last layer will be pre-trained in a supervised manner. Anyway, rest of layers are pre-trained in a unsupervised manner.

If field `input_dataset` is supplied, then `distribution` field is forbidden and, in case of pre-train supervised layer, `output_datasets` table must contain only one element.

If field `distribution` is supplied, then `input_dataset` is forbidden and, in case of pre-train supervised layer, `output_datasets` table has the same number of items than `distribution` table. In this last case, each item `output_datasets[i]` is the corresponding supervised output dataset for each item of `distribution[i].input_dataset`.

Ths table is used passed as argument to the algorithm:

```
sdae_table,deep_net = ann.autoencoders.greedy_layerwise_pretraining(params_pretrain)
```

This function returns one or two tables:

- `sdae_table = { bias={ ... }, weights={ ... } }`: which contains bias and weights of each unsupervised pre-trained layer.
- `deep_net`: An ANN component. It could be used to fine-tuning training. If you don't pre-train supervised layer, this component needs that you manually push the supervised layer.

13.2.1 Building codifier from SDAE table

```
codifier_net = ann.autoencoders.build_codifier_from_sdae_table(sdae_table,
                                    bunch_size,
                                    layers)
```

The codifier is the SDAE without the supervised layer at output. Needs the same `layers` definition as greedy pre-trainig function. Returns an ANN object which could receive a pattern as input and produces its encoding.

13.2.2 Fine-tunning supervised deep ANN

The supervised deep ANN could be fine-tuned using cross-validation training algorithm. If you pre-trained supervised layer, object `deep_net` is directly the whole ANN. Otherwise, you will need to add a new layer to the `codifier_net`, as in this example:

```
-- if you want, you could clone the deep_net to keep it as it is
local codifier_net = deep_net:clone()
codifier_net:build{ weights = deep_net:copy_weights() }
-- We add an output layer with 10 neurons and softmax activation function
local last_layer = ann.components.hyperplane{
  dot_product_weights="lastw",
  bias_weights="lastb",
  output=10
}
deep_net:push( last_layer )
deep_net:push( ann.components.actf.log_softmax()
trainer = trainable.supervised_trainer(deep_net, loss_function or nil, bunch_size or nil)
-- The output size needs to be overwitten, so it needs to be given at build method
trainer:build{ output = 10 }
weights_random = random(SEED)

-- Now, EXITS TWO WAYS to randomize the weights of last_layer
-- FIRST using the trainer
trainer:randomize_weights{
  name_match="^last[bw]$", -- the name_match is to only randomize connections which name matches
  inf=-0.1,
  sup=0.1,
  random=weights_random
}
-- SECOND using the component
-- (BE CAREFUL AND USE ONLY ONE OF THIS WAYS)
for _,cnn in pairs(last_layer:copy_weights()) do
  cnn:randomize_weights{
    inf=-0.1,
    sup=0.1,
    random=weights_random
end
```

13.2.3 Compute encoding

With a trained SDAE (without supervised layer), it is possible to compute encodings of input patterns using this function:

```
trainer = trainable.supervised_trainer(codifier_net)
encoded_dataset = trainer:use_dataset(input_dataset)
```

Chapter 14

`trainable` package

14.1 Introduction

Related with the module `require("aprilann.trainable")`

This package implements the class `trainable.supervised_trainer`, which is a powerful tool when you want to train ANNs following standard algorithms. Additionally it implements generic functions and iterators to implement training loops (dataset iterator, and training function).

If you want to do some specific tricks, it is possible to modify in your script the methods described here, or to re-implement the functionality that you need.

14.2 The supervised trainer class

The class `trainable.supervised_trainer` is the most important piece in this package. This class knows the standard API of ANN components, loss functions, optimizers, datasets, and matrix objects, so, this class use them in the correct way to train ANNs following standard algorithms.

14.2.1 Constructor

The construction of a trainer needs at least an ANN component object, but optionally, it is possible to indicate loss function, bunch size (mini-batch) and optimizer object:

```
> -- a linear component (w1 and b1 are the weight names)
> c = ann.components.hyperplane{ input=10, output=10,
                                 name                = "hyperplane",
                                 dot_product_name    = "c-w1",
                                 bias_name           = "c-b1",
                                 bias_weights        = "b1",
                                 dot_product_weights = "w1" }
> -- a trainer
> trainer = trainable.supervised_trainer(c)
```

The arguments of the constructor are positional, described here:

- ANN component: an instance of `ann.components.base()` or any sub-class of it. The trainer uses a *reference* to the given object. It is a **mandatory** argument.

- Loss function: an instance of a sub-class of `ann.loss`. The trainer uses a *reference* to the given object. If not given, it will be mandatory in `train_*` and `validate_*` methods. By default is `nil`.
- Bunch-size (mini-batch): a number with the batch size used to compute gradients. Values between 32 and 1024 are usual, depending on the optimizer and in the task. If not given, it will be mandatory in `train_*`, `validate_*`, or `use_*` methods. By default is `nil`.
- Optimizer: an instance of a sub-class of `ann.optimizer`. The trainer uses a *reference* to the given object. If not given, the default optimizer is `ann.optimizer.sgd()`.
- Smooth flag: a boolean value indicating if the gradients must be smoothed. In case of `true`, gradients will be multiplied by `1/sqrt(bunch_size+K)`, being `K` the number of times the corresponding weights matrix is shared across the ANN. By default this parameter is `true`.

```
> -- without bunch-size, without loss function,
> -- using default optimizer, smooth is true
> trainer = trainable.supervised_trainer(c)
> -- without bunch-size, using default optimizer, smooth is true
> trainer = trainable.supervised_trainer(c, ann.loss.mse())
> -- using default optimizer, smooth is true
> trainer = trainable.supervised_trainer(c, ann.loss.mse(), 64)
> -- smooth is true
> trainer = trainable.supervised_trainer(c, ann.loss.mse(), 64,
                                         ann.optimizer.rprop())
> -- all arguments given
> trainer = trainable.supervised_trainer(c, ann.loss.mse(), 64,
                                         ann.optimizer.rprop(), false)
```

14.2.2 Get and set of loss function and optimizer

14.2.2.1 set_loss_function

```
trainer:set_loss_function(loss)
```

14.2.2.2 get_loss_function

```
loss = trainer:get_loss_function()
```

14.2.2.3 set_optimizer

```
trainer:set_optimizer(optimizer)
```

14.2.2.4 get_optimizer

```
optimizer = trainer:get_optimizer()
```

14.2.3 Building the ANN

Once the `trainer` is instantiated, it is mandatory to `build` the ANN by calling the named method.

```
> trainer:build()
```

The `build` method could be called without arguments, or giving a table with the following **optional** fields:

- `input`: a number with the input size of the model. If *not given*, it will be set to the input of the given ANN component. If *given*, this size will be used as sanity check, so, it must be equal to the input size of the ANN component.
- `output`: a number with the output size of the model. Idem as previous field, but with ANN component output.
- `weights`: a dictionary (table) with strings as keys and instances of `matrix` as values. If given, the connection objects used by the components will be taken from this dictionary, so, the sizes must be equals. It could be used to initialize certain components with a given initial parameters.

```
> trainer:build{
                  input = 10, output = 10,
                  weights = {
                    -- ann.connections allocates memory for weight matrices
                    w1 = ann.connections{ input=10, output=10 },
                    b1 = ann.connections{ input=1,  output=10 },
                  }
                }
> -- giving a table, initialized using its constructor
> trainer:build{
                  input = 10, output = 10,
                  weights = {
                    w1 = ann.connections{ input=10, output=10 },
                    b1 = ann.connections{ input=1,  output=10 },
                  }
                }
```

Once the `build` method is called, it is recommended to not modify the structure of the ANN component, if you need so, after your modifications the `build` method must be called again. The connection weights **must** be modified after calling `build`, initializing them in some way. Another possibility is to deserialize a previously serialized `trainer`.

14.2.4 Serialization/deserialization

It is possible to save a `trainer` object, always in `built` state, using `util.serialize()` and `util.deserialize()` functions.

```
> util.serialize(trainer, "mytrainer-binary.net")
> trainer2 = util.deserialize("mytrainer-binary.net")
```

14.2.5 Connection weight methods

14.2.5.1 randomize_weights

`trainer:randomize_weights{ ... }`

The connection weights could be initialized randomly by using the method `trainer:randomize_weights`. This method receives a table with the following fields:

- `inf`: a number with the inferior range bound. It is a **mandatory** field.

- `sup`: a number with the superior range bound. It is a **mandatory** field.
- `random`: a `random` object. It is a **mandatory** field.
- `use_fanin`: a boolean value indicating if apply a factor fan-in of each layer for its initialization. It is an **optional** field. By default is `false`.
- `use_fanout`: a boolean value indicating if apply a factor fan-out of each layer for its initialization. It is an **optional** field. By default is `false`
- `name_match`: a string with a Lua pattern used to filter which connection objects will be initialized. It is an **optional** field, by default is `.*`.

The weights will be initilized in the range `[c*inf, c*sup]` where `c` is a factor which depends on `use_fanin` and `use_fanout` arguments:

- If none given, then `c=1` for all weight layers.
- If given **only** `use_fanin=true`, then `c` is computed depending in the fan-in of each layer, being `c = 1/sqrt(fanin)`.
- If given **only** `use_fanout=true`, then `c` is computed depending in the fan-out of each layer, being `c = 1/sqrt(fanout)`.
- If both given, `use_fanin=true` and `use_fanout=true`, then `c` is computed depending in the fan-in and fan-out of each layer, being `c = 1/sqrt(fanin + fanout)`.

```
> -- initilize only bias weights
> trainer:randomize_weights{
                             inf = -0.1,
                             sup =  0.1,
                             random = random(213924),
                             name_match = "b.*",
                           }
> -- initilize only non-bias weights
> trainer:randomize_weights{
                             inf = -0.1,
                             sup =  0.1,
                             random = random(213924),
                             name_match = "w.*",
                             use_fanin   = true,
                             use_fanout  = true,
                           }
> -- initilize all the weights
> trainer:randomize_weights{
                             inf = -0.1,
                             sup =  0.1,
                             random = random(213924),
                           }
```

Once the `trainer` is built, it is possible to do some introspection in order to modify or execute methods of connection weights.

14.2.5.2 count_weights

```
number = trainer:count_weights( [pattern=.*] )
```

This method returns the number of connection weights in the current `trainer`. **Optionally** the method receives a Lua pattern filtering the counting process to only the weights whom name matches the pattern.

```
> = trainer:count_weights()
2
```

14.2.5.3 weights

```
number = trainer:weights(name)
```

This method returns the `matrix` object with the given `name`.

```
> w1 = trainer:weights("w1")
> = type(w1)
matrix
> b1 = trainer:weights("b1")
> = type(b1)
matrix
```

14.2.5.4 get_components_of

```
table = trainer:get_components_of(weights_name)
```

This method returns a table with `ann.components` objects which share the given `weights_name` connection weights. It returns a table because a connection weights object could be shared by more than one ANN components.

```
> iterator(ipairs( trainer:get_components_of("w1") )):apply(print)
1   instance 0xfcf5b0 of ann.components.base
> iterator(ipairs( trainer:get_components_of("b1") )):apply(print)
1   instance 0xff3c90 of ann.components.base
```

14.2.5.5 get_weights_table

```
table = trainer:get_weights_table()
```

This method returns a dictionary (table) with all the connection weights. This dictionary has the same structure as the `weights` field of `trainer:build(...)` method.

```
> weights_table = trainer:get_weights_table()
> print(weights_table)
table: 0x1310a00
```

14.2.5.6 ... = trainer:iterate_weights([pattern=.*])

This method returns a Lua iterator function which iterates over all the connection weights which name matches the given `pattern`.

```
> for cnn_name,cnn in trainer:iterate_weights() do print(cnn_name) end
b1
w1
> iterator( trainer:iterate_weights("w.*") ):select(1):apply(print)
w1
```

14.2.5.7 norm2

```
number = trainer:norm2([pattern=.*])
```

This method computes the 2-norm of the connection weight objects whom name matches the given pattern.

```
> = trainer:norm2()
0.24416591227055
> = trainer:norm2("b.")
0
> = trainer:norm2("w.")
0.24416591227055
```

14.2.5.8 size

```
number = trainer:size()
```

This methods returns the number of parameters (weights) in the current ANN component.

```
> = trainer:size()
110
```

14.2.5.9 Example

By using these methods it is possible to manipulate by-hand the connection weights, as in the following example, which initializes to zero the bias connections:

```
> for _,cnn in trainer:iterate_weights("b.*") do cnn:zeros() end
```

REMEMBER that connection weight objects are `matrix` instances.

The following code shows at screen all the weight matrices:

```
> iterator(trainer:iterate_weights()):apply(print)
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
# Matrix of size [10,1] [0x10aca60 data= 0x10acb20]
 0.0575503   0.0516265  -0.0306808   0.035404    0.0118243  ...
```

```
 0.0281929   0.0877731   0.0842627  -0.0379949  -0.091877   ...
-0.0332023   0.0576623  -0.0335078   0.0251189  -0.0578111  ...
-0.0335119   0.0162495  -0.00910386 -0.0949801   0.00303258 ...
-0.0361652  -0.0389352   0.0628194  -0.0622919  -0.0206459  ...
 0.0583717   0.0910834  -0.0889903  -0.0142328  -0.0750175  ...
-0.0895628   0.0412171   0.0308301  -0.0680314   0.0948681  ...
-0.00439932  0.0975324   0.00736945  0.013484   -0.079681   ...
-0.0859327   0.0332012   0.0374489  -0.0555631  -0.0308727  ...
 0.0375495  -0.0474079   0.0450424  -0.0822513  -0.00803252 ...
# Matrix of size [10,10] [0x10fe150 data= 0x10fe210]
```

14.2.6 Component methods

Once the `trainer` is built, it is also possible to do introspection for getting or modify ANN components.

14.2.6.1 count_components

```
number = trainer:count_components( [pattern=.*] )
```

This method returns the number of ANN components in the current `trainer`. **Optionally** the method receives a Lua pattern filtering the counting process to only the components whom name matches the pattern.

```
> = trainer:count_components()
3
```

14.2.6.2 get_components_of

```
object = trainer:get_component()
```

This method returns the *root* ANN component, which was given to the constructor.

```
> = trainer:get_component()
instance 0x1175430 of ann.components.hyperplane
```

14.2.6.3 component

```
number = trainer:component(name)
```

This method returns an ANN component object given its `name`.

```
> = trainer:component("hyperplane")
instance 0x1175430 of ann.components.base
```

14.2.6.4 get_weights_of

```
object = trainer:get_weights_of(component_name)
```

This method returns the `matrix` object which belongs to the given `component_name`.

```
> = trainer:get_weights_of("hyperplane")
nil
> = type( trainer:get_weights_of("c-w1") )
matrix
> = type( trainer:get_weights_of("c-b1") )
matrix
```

14.2.6.5 iterate_components

```
...  = trainer:iterate_components( [pattern=.*] )
```

This method returns a Lua iterator function which iterates over all the ANN components which name matches the given `pattern`.

```
> for name,c in trainer:iterate_components() do print(name,c) end
hyperplane  instance 0x1175430 of ann.components.base
c-w1    instance 0xfcf5b0 of ann.components.base
c-b1    instance 0xff3c90 of ann.components.base
> iterator( trainer:iterate_components("c-w.*") ):apply(print)
c-w1    instance 0xfcf5b0 of ann.components.base
```

14.2.7 Optimizer methods

The following methods are shortcuts to modify hyperparameters of the `optimizer` object.

14.2.7.1 has_option

```
desc = trainer:has_option(option)
```

Returns the description of the given `option` if it exists at the `optimizer`. Otherwise it returns `nil`.

14.2.7.2 set_option

```
trainer:set_option(option, value)
```

Sets the given `option` name to the given `value`. Throws an error in case the option doesn't exists at the `optimizer`.

14.2.7.3 get_option

```
value = trainer:get_option(option)
```

Returns the value of a given `option` name, or throws an error if the option doesn't exits at the `optimizer`.

14.2.7.4 set_layerwise_option

```
trainer:set_layerwise_option(pattern, option, value)
```

This method needs that the `trainer` was in **built** state. It traverses all the connection weight objects which name matches the given `pattern` string, and sets its layer-wise `option` name to the given `value`.

14.2.7.5 get_option_of

```
value = trainer:get_option_of(name, option)
```

This method returns the `option` value which applies to the given connection weight object `name`.

14.2.8 One batch training, validation and gradients check

The following methods are prepared to work with a bunch of patterns (mini-batch). They do one batch step of the algorithms, and could be rewritten to do specific things.

14.2.8.1 train_step

```
mu, matrix = trainer:train_step(input, target, loss, optimizer)
```

This method executes one training step, using the given data:

- `input` is a `token` with a bunch (mini-batch) of data, usually it is a `matrix` instance, where rows are patterns and columns features.
- `target` is a `token` with a bunch (mini-batch) of data. Usually it is a `matrix` instance.
- `loss` is a `ann.loss` function object. It is **optional**, if not given it uses the loss function object instantiated at the `trainer` object.
- `optimizer` is an `ann.optimizer` object. It is **optional**, if not given it uses the optimizer object instantiated at the `trainer` object.

The method returns two values:

- `mu` is the mean of the loss function at the given batch of patterns.
- `matrix` is a one-dimensional matrix with the loss of every pattern.

```
> mean,loss_matrix = trainer:train_step(input, target)
```

14.2.8.2 validate_step

```
mu, matrix = trainer:validate_step(input, target, loss)
```

This method executes one validate step, using the given data:

- `input` is a `token` with a bunch (mini-batch) of data. It can be a `matrix` instance.
- `target` is a `token` with a bunch (mini-batch) of data. It can be a `matrix` instance.
- `loss` is a `ann.loss` function object. It is **optional**, if not given it uses the loss function object instantiated at the `trainer` object.

The method returns two values:

- `mu` is the mean of the loss function at the given batch of patterns.
- `matrix` is a one-dimensional matrix with the loss of every pattern.

The validate step evaluates the performance of the ANN component using the given loss function, but it doesn't train the parameters.

```
> mean,loss_matrix = trainer:validate_step(input, target)
```

14.2.8.3 compute_gradients_step

```
g, mu, mat = trainer:compute_gradients_step(i, t, l [, g ])
```

This method compute the gradients of the given data, but doesn't train the parameters. The given gradients could be used to perform a manual training or gradient checking.

The arguments are:

- `input` is a `token` with a bunch (mini-batch) of data. It is usually a `matrix` instance.
- `target` is a `token` with a bunch (mini-batch) of data. It is usually a `matrix` instance.
- `loss` is a `ann.loss` function object. It is **optional**, if not given it uses the loss function object instantiated at the `trainer` object.
- `gradients` is a dictionary with the gradient matrices of every connection weights object. It is **optional**, if not given, the matrices will be allocated, if given, the allocation could be avoided.

The method returns three values:

- `gradients` the gradients dictionary.
- `mu` is the mean of the loss function at the given batch of patterns.
- `matrix` is a one-dimensional matrix with the loss of every pattern.

```
> gradients,mean,loss_mat = trainer:compute_gradients_step(input, target)
```

14.2.8.4 grad_check_step

```
boolean = trainer:grad_check_step(i, t [, boolean [, loss ] ] )
```

This method compute the gradients of the given data, and executes a gradient checking algorithm using numerical differentiation. The arguments are:

- `input` is a `token` with a bunch (mini-batch) of data. It is usually a `matrix` instance.
- `target` is a `token` with a bunch (mini-batch) of data. It is usually a `matrix` instance.
- `boolean`, if `true`, it indicates high verbosity. It is **optional**, by default is `false`.
- `loss` is a `ann.loss` function object. It is **optional**, if not given it uses the loss function object instantiated at the `trainer` object.

The method returns a boolean indicating if the gradient checking algorithm success or fails.

```
> trainer:grad_check_step(input, target) or error("Gradients checking fails")
```

14.2.9 Dataset methods training, validation and gradients check

This methods perform a traversal over a dataset with a large number of patterns, training or evaluating the model. The dataset is divided into batches of `bunch_size` size.

14.2.9.1 train_dataset

```
loss_mu,loss_var = trainer:train_dataset{ ...  }
```

This method is used to train by using a given dataset. Different training schedules are possible, depending on the parameters given to the method. In any case, this methods return two values:

- The mean of the loss function over all the patterns.
- The variance of the loss function over all the patterns.

The following training schedules are available:

- **Training with all the patterns, in sequential way**: the fields of the given table are:
  - `input_dataset` a `dataset` with the data for input of ANN components.
  - `output_dataset` a `dataset` with the data for target outputs of ANN components (supervision).
  - `bunch_size` the mini-batch size, **optional** parameter. If not given, the bunch size instantiated at the `trainer` will be used.
  - `loss` a loss function object, **optional** parameter. If not given, the loss instantiated at the `trainer` will be used.
  - `optimizer` an optimizer object, **optional** parameter. If not given, the optimizer instantiated at the `trainer` will be used.
- **Training with all the patterns in shuffled way**: the fields of the given table are:
  - `input_dataset` a `dataset` with the data for input of ANN components.
  - `output_dataset` a `dataset` with the data for target outputs of ANN components (supervision).
  - `random` a `random` object instance, used to shuffle the patterns.
  - `bunch_size` the mini-batch size, **optional** parameter. If not given, the bunch size instantiated at the `trainer` will be used.
  - `loss` a loss function object, **optional** parameter. If not given, the loss instantiated at the `trainer` will be used.
  - `optimizer` an optimizer object, **optional** parameter. If not given, the optimizer instantiated at the `trainer` will be used.
- **Training with replacement**: the fields of the given table are:
  - `input_dataset` a `dataset` with the data for input of ANN components.
  - `output_dataset` a `dataset` with the data for target outputs of ANN components (supervision).
  - `random` a `random` object instance, used to shuffle the patterns.
  - `replacement` a given number with the size of the replacement.
  - `bunch_size` the mini-batch size, **optional** parameter. If not given, the bunch size instantiated at the `trainer` will be used.
  - `loss` a loss function object, **optional** parameter. If not given, the loss instantiated at the `trainer` will be used.
  - `optimizer` an optimizer object, **optional** parameter. If not given, the optimizer instantiated at the `trainer` will be used.
- **Training with distribution**: the fields of the given table are:
  - `distribution` is an array of tables, where each table contains:
    - `input_dataset` a `dataset` with the data for input of ANN components.
    - `output_dataset` a `dataset` with the data for target outputs of ANN components (supervision).
    - `prob` a number with the probability of taken a pattern from this data source.
  - `random` a `random` object instance, used to shuffle the patterns.

  - `replacement` a given number with the size of the replacement.
  - `bunch_size` the mini-batch size, **optional** parameter. If not given, the bunch size instantiated at the `trainer` will be used.
  - `loss` a loss function object, **optional** parameter. If not given, the loss instantiated at the `trainer` will be used.
  - `optimizer` an optimizer object, **optional** parameter. If not given, the optimizer instantiated at the `trainer` will be used.

14.2.9.2 validate_dataset

```
loss_mu,loss_var = trainer:validate_dataset{ ...  }
```

This method is used to validate the model by using a given dataset. Different validation schedules are possible, depending on the parameters given to the method. In any case, this method returns two values:

- The mean of the loss function over all the patterns.
- The variance of the loss function over all the patterns.

The following validation schedules are available:

- **Validate with all the patterns, in sequential way**: the fields of the given table are:
  - `input_dataset` a `dataset` with the data for input of ANN components.
  - `output_dataset` a `dataset` with the data for target outputs of ANN components (supervision).
  - `bunch_size` the mini-batch size, **optional** parameter. If not given, the bunch size instantiated at the `trainer` will be used.
  - `loss` a loss function object, **optional** parameter. If not given, the loss instantiated at the `trainer` will be used.
- **Validate with replacement**: the fields of the given table are:
  - `input_dataset` a `dataset` with the data for input of ANN components.
  - `output_dataset` a `dataset` with the data for target outputs of ANN components (supervision).
  - `random` a `random` object instance, used to shuffle the patterns.
  - `replacement` a given number with the size of the replacement.
  - `bunch_size` the mini-batch size, **optional** parameter. If not given, the bunch size instantiated at the `trainer` will be used.
  - `loss` a loss function object, **optional** parameter. If not given, the loss instantiated at the `trainer` will be used.

14.2.9.3 use_dataset

```
output_ds = trainer:use_dataset{ ...  }
```

This method receives a table with a one or two datasets and computes the output of the ANN component for every pattern. **Note** that this method has a large use of memory, because it needs a dataset where to store the ANN output for every pattern. *Please, be careful when using it.*

It receives a table with fields:

- `input_dataset` a `dataset` with the input data for the ANN component.
- `output_dataset` a `dataset` with enough space to store the output of the ANN component for every pattern in the `input_dataset`. If not given, the `output_dataset` will be allocated automatically with the required size.

This method returns the `output_dataset` with the produced data.

14.2.9.4 grad_check_dataset

```
boolean = trainer:grad_check_dataset{ ...  }
```

14.3 Training function loop classes

This two classes are useful to build a training loop with a default stopping criterion.

14.3.1 train_wo_validation

`trainable.train_wo_validation` class

This class implements the training function without validation, using a stopping criterion based on percentage of improvement in training or in number of epochs. The following **methods** are defined.

14.3.1.1 Constructor

```
train_func = trainable.train_wo_validation{ ...  }
```

The constructor, which receives a table with the following fields:

- `min_epochs=1`: the minimum number of epochs of the training. It is **optional**.
- `max_epochs`: the maximum number of epochs of the training, if `min_epochs==max_epochs` then stopping criteria will be number of epochs.
- `percentage_stopping_criterion=0.01`: a number in range `[0,1]` indicating the threshold for the percentage of improvement in training loss between two consecutive epochs. If the train loss improvement is less than this number, the training will stops. It is an **optional** field.
- `first_epoch=1`: indicates the number of the first epoch. It is **optional**.

```
> -- instance using percentage_stopping_criterion=0.01 (default value)
> train_func = trainable.train_wo_validation{ max_epochs = 100 }
```

14.3.1.2 execute

```
boolean = train_func:execute(epoch_function)
```

This method executes one epoch step. It is the most important method. It receives an *epoch function*, which is a closure with the responsibility of perform training with one epoch, and it must returns two values: the trained *model* and the *training* loss. The method returns `true` or `false` depending in if the stopping criterion is satisfied or not.

```
> while train_func:execute(function()
                local tr = trainer:train_dataset(datosentrenar)
                return trainer,tr
              end) do
  print(train_func:get_state_string())
end
```

14.3.1.3 set_param

```
train_func:set_param(name,value)
```

This method modifies the value of a parameter previously given at the constructor (or used with its default value).

14.3.1.4 get_param

```
value = train_func:get_param(name)
```

This method returns the value of a parameter previously given at the constructor (or used with its default value).

14.3.1.5 get_state

```
epoch,tr_loss,tr_improvement,last=train_func:get_state()
```

This method returns the internal state of the object. `last` is the trained model returned by the last call to *epoch function*.

14.3.1.6 get_state_table

```
state = train_func:get_state_table()
```

This method returns a table with the following fields:

- `state.current_epoch`: the current epoch.
- `state.train_error`: the train loss at last epoch.
- `state.train_improvement`: the train loss relative improvement.
- `state.last`: the trained model returned by the last call to *epoch function*.

14.3.1.7 get_state_string

```
string = train_func:get_state_string()
```

This method returns a string for printing purposes, with the following format:

```
string.format("%5d %.6f    %.6f",
              state.current_epoch,
              state.train_error,
              state.train_improvement)
```

14.3.1.8 Code example

Finally, here is a code example showing how to use this class:

```
> trainer    = trainable.supervised_trainer(thenet, ann.loss.mse(), 64)
> train_func = trainable.train_wo_validation{ max_epochs = 100 }
> while train_func:execute(function()
              local tr = trainer:train_dataset(training_data)
              return trainer,tr
            end) do
  print(train_func:get_state_string())
  util.serialize({ train_func, training_data.shuffle }, "training.lua")
end
```

The following is an example of loading previously saved object:

```
> train_func,shuffle    = util.deserialize("training.lua")
> trainer               = train_func:get_state_table().last
> thenet                = trainer:get_component()
> training_data.shuffle = shuffle
> while train_func:execute(function()
              local tr = trainer:train_dataset(training_data)
              return trainer,tr
            end) do
  print(train_func:get_state_string())
  util.serialize({ train_func, training_data.shuffle }, "training.lua")
end
```

14.3.2 train_holdout_validation

`trainable.train_holdout_validation` class

This class implements the training function with a holdout validation set, using a stopping criterion based on validation or error in number of epochs. This object follows the Pocket Algorithm, so, it keeps the model which has the best validation loss during the training. A tolerance in the relative error could be used to decided a minimum improvement to take the model as the best. The following **methods** are defined.

14.3.2.1 Constructor

`train_func = trainable.train_holdout_validation{ ... }`

The constructor, which receives a table with the following fields:

- `min_epochs=1`: the minimum number of epochs of the training. It is **optional**.
- `max_epochs`: the maximum number of epochs of the training, if `min_epochs==max_epochs` then stopping criteria will be number of epochs.
- `epochs_wo_validation=0`: the number of epochs where validation loss is ignored, so the best model is the last given. It is **optional**.
- `stopping_criterion=function() return false end`: a stopping criterion function. It will be used to determine when the training must be stopped. The given function is called given it a table with the output of `get_state_table()` method. It is **optional**. Basic criterion functions are defined in `trainable` table, and described below this section.
- `first_epoch=1`: indicates the number of the first epoch. It is **optional**.

- `tolerance=0`: the `tolerance>=0` is the minimum relative difference to take the current validation loss as the best. It is **optional**.

```
> criterion  = trainable.stopping_criteria.make_max_epochs_wo_imp_relative(2)
> train_func = trainable.train_holdout_validation{
                 stopping_criterion = criterion,
                 max_epochs         = max_epochs
               }
```

14.3.2.2 execute

```
boolean = train_func:execute(epoch_function)
```

This method executes one epoch step. It is the most important method. It receives an *epoch function*, which is a closure with the responsibility of perform training with one epoch, and it must returns three values: the trained *model*, the *training* loss and the *validation* loss. The method returns `true` or `false` depending in if the stopping criterion is satisfied or not.

```
> while train_func:execute(function()
               local tr = trainer:train_dataset(training_data)
               local va = trainer:validate_dataset(validation_data)
               return trainer,tr,va
             end) do
  print(train_func:get_state_string())
  local state = train_func:get_state_table()
  if state.best_epoch == state.current_epoch then
    util.serialize({ train_func, training_data.shuffle }, "training.lua")
  end
end
```

14.3.2.3 set_param

```
train_func:set_param(name,value)
```

This method modifies the value of a parameter previously given at the constructor (or used with its default value).

14.3.2.4 get_param

```
value = train_func:get_param(name)
```

This method returns the value of a parameter previously given at the constructor (or used with its default value).

14.3.2.5 get_state

```
epoch,tr_loss,va_loss,...=train_func:get_state()
```

This method returns the internal state of the object: epoch, training loss, validation loss, best epoch, best validation loss, best model clone, last given model.

14.3.2.6 get_state_table

`state = train_func:get_state_table()`

This method returns a table with the following fields:

- `state.current_epoch`: the current epoch.
- `state.train_error`: the train loss at last epoch.
- `state.validation_error`: the validation loss at last epoch.
- `state.best_epoch`: the epoch where the best validation loss where found.
- `state.best_val_error`: the validation loss at the best epoch.
- `state.best`: the trained model which achieves the best validation error.
- `state.last`: the trained model returned by the last call to *epoch function*.

14.3.2.7 get_state_string

`string = train_func:get_state_string()`

This method returns a string for printing purposes, with the following format:

```
  string.format("%5d %.6f %.6f    %5d %.6f",
                state.current_epoch,
                state.train_error,
                state.validation_error,
                state.best_epoch,
                state.best_val_error)
```

14.3.2.8 Code example

Finally, here is a code example showing how to use this class:

```
> trainer    = trainable.supervised_trainer(thenet, ann.loss.mse(), 64)
> criterion  = trainable.stopping_criteria.make_max_epochs_wo_imp_relative(2)
> train_func = trainable.train_holdout_validation{
                 stopping_criterion = criterion,
                 max_epochs         = max_epochs
               }
> while train_func:execute(function()
               local tr = trainer:train_dataset(training_data)
               local va = trainer:validate_dataset(validation_data)
               return trainer,tr,va
             end) do
  print(train_func:get_state_string())
  local state = train_func:get_state_table()
  if state.best_epoch == state.current_epoch then
    util.serialize({ train_func, training_data.shuffle }, "training.lua")
  end
end
```

14.4 Stopping criteria

For holdout-validation scheme, exists two predefined stopping criteria, which are function builders (they return the function used as criterion):

- `trainable.stopping_criteria.make_max_epochs_wo_imp_absolute`: which receives a constant indicating the maximum number of epochs without improve validation. A tipical value is between 10 and 20, depending in the task.
- `trainable.stopping_criteria.make_max_epochs_wo_imp_relative`: which receives a constant indicating the maximum value for current_epoch/best_epoch. A tipical value for this is 2.

This two criteria could be used as this:

```
train_func = trainable.train_holdout_validation{
  ...
  stopping_criterion = trainable.stopping_criteria.make_max_epochs_wo_imp_relative(2),
  ...
}
```

Also you can create your own stopping criterion, which is a function which receives a table:

```
train_func = trainable.train_holdout_validation{
  ...
  stopping_criterion = function(t)
    -- t contains this fields:
    --   * current_epoch
    --   * best_epoch
    --   * best_val_error
    --   * train_error
    --   * validation_error
    return true IF STOPPING CRITERIA(t) IS TRUE
  end,
  ...
}
```

14.5 Dataset iterator functions

The class `trainable.supervised_trainer` uses some generic dataset iterator functions, available for the user if needed. Two functions are available: `trainable.dataset_pair_iterator` and `trainable.dataset_multiple_iterator`. The first is a wrapper around the second one. This iterators could perform different traverse methods, depending in the given parameters.

14.5.1 dataset_pair_iterator

Lua iterator = `trainable.dataset_pair_iterator{ ... }`

This iterator performs a synchronized traversal of two given datasets (normally it is a pair input/output). The function returns a Lua iterator which returns three values every time it is called: `input` pattern (a `token`, usually a `matrix` instance), `output` pattern (a `token`, usually a `matrix` instance), and a Lua table with the indexes of the patterns taken in the bunch.

The available traversal modes are:

- **Traverse all the patterns, in sequential way**: the fields of the given table are:
  - `input_dataset` a `dataset` with the data for input of ANN components.
  - `output_dataset` a `dataset` with the data for target outputs of ANN components (supervision).
  - `bunch_size` the mini-batch size.
- **Traverse all the patterns in shuffled way**: the fields of the given table are:
  - `input_dataset` a `dataset` with the data for input of ANN components.
  - `output_dataset` a `dataset` with the data for target outputs of ANN components (supervision).
  - `shuffle` a `random` object instance, used to shuffle the patterns.
  - `bunch_size` the mini-batch size.
- **Traverse with replacement**: the fields of the given table are:
  - `input_dataset` a `dataset` with the data for input of ANN components.
  - `output_dataset` a `dataset` with the data for target outputs of ANN components (supervision).
  - `shuffle` a `random` object instance, used to shuffle the patterns.
  - `replacement` a given number with the size of the replacement.
  - `bunch_size` the mini-batch size.
- **Traverse with distribution**: the fields of the given table are:
  - `distribution` is an array of tables, where each table contains:
    - `input_dataset` a `dataset` with the data for input of ANN components.
    - `output_dataset` a `dataset` with the data for target outputs of ANN components (supervision).
    - `prob` a number with the probability of taken a pattern from this data source.
  - `shuffle` a `random` object instance, used to shuffle the patterns.
  - `replacement` a given number with the size of the replacement.
  - `bunch_size` the mini-batch size.

```
> ds_params = { input_dataset = my_input_ds, output_dataset = my_output_ds }
> for input,output,idxs in trainable.dataset_pair_iterator(ds_params) do
    -- you can give the input/output to an ANN and loss function
    print(input,output,idxs)
  end
```

14.5.2 dataset_multiple_iterator

Lua iterator = `trainable.dataset_multiple_iterator{ ... }`

This iterator performs a synchronized traversal of any number given datasets (normally it is a pair input/output). The function returns a Lua iterator which returns as many values as the number of given datasets plus one: one pattern for each dataset, plus a Lua table with the indexes of the patterns taken in the bunch.

The available traversal modes are:

- **Traverse all the patterns, in sequential way**: the fields of the given table are:
  - `datasets`: a Lua table with the list of dataset for traversal.
  - `bunch_size` the mini-batch size.
- **Traverse all the patterns in shuffled way**: the fields of the given table are:
  - `datasets`: a Lua table with the list of dataset for traversal.
  - `shuffle` a `random` object instance, used to shuffle the patterns.

  - `bunch_size` the mini-batch size.

- **Traverse with replacement**: the fields of the given table are:
  - `datasets`: a Lua table with the list of dataset for traversal.
  - `shuffle` a `random` object instance, used to shuffle the patterns.
  - `replacement` a given number with the size of the replacement.
  - `bunch_size` the mini-batch size.

- **Traverse with distribution**: the fields of the given table are:
  - `distribution` is an array of tables, where each table contains:
    * `datasets`: a Lua table with the list of dataset for traversal.
    * `prob` a number with the probability of taken a pattern from this data source.
  - `shuffle` a `random` object instance, used to shuffle the patterns.
  - `replacement` a given number with the size of the replacement.
  - `bunch_size` the mini-batch size.

```
> ds_params = { datasets = { my_ds1, my_ds2, my_ds3 } }
> for token1,token2,token3,idxs in trainable.dataset_multiple_iterator(ds_params) do
    -- you can give the token1,token2,token3 to an ANN or a loss function
    print(token1,token2,token3,idxs)
  end
```

Chapter 15

`random` package

15.1 Introduction

Package `random` could be loaded via the standalone binary, or in Lua with `require("aprilann.random)`.

The `random` class is useful to generate pseudo-random numbers, and is widely used by ANN components and other classes of APRIL-ANN. It is based on Mersenne Twister, basically it is a binding of the original C++ code of Mersenne Twister.

15.2 Methods of random class

`random` contains the following methods:

15.2.1 Constructor

`obj = random( [seed] )`

A constructor of the object. The parameter is optional, if not given, it is taken from the current time of the machine. If given, it could be:

- a seed `number` for the initialization of the random generator;
- a `table` with seeds for the initialization of the random generator.

15.2.2 rand

`number = obj:rand( [number] )`

Returns a double random number in the interval `[0,n]`, being `n` the given parameter. If not given any parameter, by default `n=1`.

15.2.3 randExc

`number = obj:randExc( [number] )`

Returns a double random number in the interval `[0,n)`, being `n` the given parameter. If not given any parameter, by default `n=1`.

15.2.4 randDblExc

```
number = obj:randDblExc( [number] )
```

Returns a double random number in the interval `(0,n)`, being `n` the given parameter. If not given any parameter, by default `n=1`.

15.2.5 randInt

```
number = obj:randInt( [x, [ y ] ] )
```

Returns an integer random number in the interval `[x,y]`. If only one argument is given, then the interval will be `[0,x]`. If zero argument are given, the interval will be `[0,2^32-1]`.

15.2.6 shuffle

```
table = obj:shuffle(N)
```

Returns a table with size `N`, which is a permutation of the indices of an N-sized array.

```
table = obj:shuffle(table)
```

Returns a random permutation of the given table array.

15.2.7 choose

```
number = obj:choose(size)
```

$eturns a random element for an array of the given `size`. It is equivalent to `obj:randInt(1,size)`.

15.2.8 randNorm

```
number = obj:randNorm(mean,variance)
```

Returns a random number sampled from a Gaussian with the given `mean` and `variance` parameters.

15.2.9 seed

```
obj:seed(number)
```

Modifies the seed, see the constructor.

```
obj:seed(table)
```

Modifies the seed, see the constructor.

15.2.10 clone

```
another = obj:clone()
```

Returns a *deep copy* of the caller object.

Chapter 16

autodiff package

16.1 Introduction

Related with package `require("aprilann.autodiff")`.

The `autodiff` package is an **in-development** package, which adds automatic differentiation to APRIL-ANN toolkit. It is inspired into Theano, but, in this case, using Lua and APRIL-ANN matrix library instead of tensors in Python.

This packages works over three main data types: constants, scalars, matrices. For example, the addition of two scalars will need the following code:

```
> AD = autodiff -- for simplicity
> a,b = AD.scalar('a b')
> c = a + b
> = c
(+ a b)
```

Every expression is stored as a graph of dependencies between operations and its arguments. Every symbol (like `a`, `b` or `c`) is a special Lua table. Symbols has a name, in the case of ground symbols (like `a` or `b`), its name is the symbol itself. In the case of operations (like `c`), the name is what you see when you do `print(c)`.

```
> = a.name,b.name,c.name
a   b   (+ a b)
```

A symbol only exists once, so, if you declare another symbol with the same name, or the same operation, it will be a reference to the first symbol declaration. They are exactly the same variable. In this way, the computation is performed in a symbolic graph, and the use of *memoization* allows to ensures that every operation is only computed once, even if it is repeated several times.

However, depending in the declaration order, the same operation could be represented in different ways. It is possible to optimize the graph applying basic arithmetic properties.

```
> d = c + b + a
> = d
(+ (+ (+ a b) b) a)
> e = AD.optimize(d)
> = e
(+ (* 2 a) (* 2 b))
```

It is possible to operate with Lua numbers, which are automatically coerced into `autodiff.constant` symbol:

```
> f = 4*b*a/b + 2
> = f
(+ (* (* (* 4 b) a) (^ b -1)) 2)
> = AD.optimize(f)
(+ (* 4 a) 2)
```

Math operators are overloaded using Lua metamethods, so it is possible to do use math operators in a standard way: addition, subtraction, multiplication, division, power, unary minus. Nevertheless, this operations are defined as functions in `autodiff` package.

```
> op = AD.op    -- for simplicity
> c = AD.op.div( AD.op.add(a,b), 4)
> = c
(* (+ a b) 0.25)
```

Constants are automatically folded, performing constant operations. Subtraction is transformed in addition with multiplication by -1. Division is transformed into multiplication with power by -1. This reduces a lot the simplification effort.

```
> c = a/b - a
(+ (* (^ b -1) a) (* -1 a))
```

But this transformations makes difficult to follow the printed string. It is possible to produce a graph in dot format.

```
> AD.dot_graph(c, "mygraph.dot")
```

The most interesting thing is the *automatic differentiation* procedure, which receives a list of variables w.r.t differentiate. The function returns a variable number of arguments, one graph for each differentiated variable.

```
> c = a * b
> da,db = AD.diff(c, {a,b})
> = da
b
> = db
a
```

Any graph could be evaluated with a given instantiation of its internal ground variables.

```
> c = a * b
> = c:eval{ a=5, b=2 }
10
> = da:eval{ a=3, b=2 }
2
> = db:eval{ a=3, b=2 }
3
```

Additionally, it is possible to **compile** several graphs together, sharing the computation of equal operations. The compilation procedure automatically applies the optimization function to the given graphs. It receives the list of graphs (symbols) which you want to compile together, and a list of input variables (in order). Optionally it is possible to give a dictionary Lua table with values which will be shared between the compiled function and the caller Lua virtual machine.

```
> -- AD.func does the compilation
> my_func1 = AD.func({ c, da, db }, { a,b })
> = my_func1(2,5)
10    5     2
> shared = { b=4 }
> my_func2 = AD.func({ c, da, db }, { a }, shared)
> = my_func2(2) -- 2 * 4
8     4     2
> shared.b = 10
> = my_func2(2) -- 2 * 10
20    10    2
```

16.2 Basic functionality

The most important concept is the *symbol*, which could be a ground variable (with a declared type) or a computational graph. Every symbol has a name which identifies it univocally. Every symbol has overloaded basic math operations (addition, subtraction, multiplication, division, unary minus, power), allowing to use standard math symbols to produce complex operations. Computational graph symbols represent operations. Every symbol implements a generic interface. Depending in the type, symbols are constructed in different ways:

```
> AD  = autodiff -- for simplicity
> a,b = AD.scalar('a b') -- scalar constructor
> c   = a + 4*b          -- 4 is casted to AD.constant type
> m   = AD.matrix('m')   -- matrix constructor
> d   = m * c
```

The type of computational graph symbols (operations) depends in the arguments of the operation. Inference rules are declared in order to decide when an operation is over matrices, scalar, constants, or any of the possible symbol types.

16.2.1 eval

```
value = s:eval( table [, cache] )
```

Evaluates the caller symbol by using the given table of ground variables. All the ground variables need to be defined (except constants).

```
> = d:eval{ m=matrix(3,3):linear(), a=10, b=2 }
 0          18         36
 54         72         90
 108        126        144
# Matrix of size [3,3] in row_major [0x2b6cd60 data= 0x293e4a0]
```

An **optional** table could be given as second argument, which will be used as *cache* table (for memoization), allowing to share computations between different symbols evaluation. In this case, it is not necessary to give all the ground variables, as far as all the needed partial operations where computed and moemoized in the given cache table.

```
> cache = {}
> = c:eval({ a=10, b=2 }, cache) -- computes c=a + 4*b
18
> = d:eval({ m=matrix(3,3):linear() }, cache) -- computes d = m * 18
 0           18          36
 54          72          90
 108         126         144
# Matrix of size [3,3] in row_major [0x2b6cd60 data= 0x293e4a0]
```

16.2.2 func

`function = autodiff.func(symbols, args, shared [, optimize=true])`

This function compiles together the given set of symbols, producing as result a Lua function which receives as arguments the ground variables indicated in the `args` table (ordered), and shares the given table of `shared` ground variables. Additionally, an **optional** fourth argument could be given, indicating with a `boolean` if optimization needs to be performed or not. If not given, this fourth argument will be considered `true`.

The symbols list (first argument) could be a table of symbols or a symbol. The returned Lua function will receive as many arguments as the size of `args` table array, and will produce as many values as size of `symbols` list.

```
> f = AD.func(d, {a,b}, {m = matrix(3,3):linear()})
> = f(10,2)
 0           18          36
 54          72          90
 108         126         144
# Matrix of size [3,3] in row_major [0x1cddcd0 data= 0x1c86b20]
```

16.2.3 diff

`ds1,ds2,...  = autodiff.diff(symbol, { s1, s2, ...  } [, seed] )`

This function differentiates the given symbol w.r.t. all the given array of ground variables (second argument). It returns multiple outputs, as many as the size of the ground variables array.

```
> dm,da = autodiff.diff(d, { m, a })
> = dm
(.* (+ (* 4 b) a) (fill (.* m (+ (* 4 b) a)) 1))
> = da
(.* m (fill (.* m (+ (* 4 b) a)) 1))
> = d
(.* m (+ (* 4 b) a))
```

Looking to the produced computations, they have one thing in common: both uses a *fill* operation which receives as input the original `d` computational graph and the number `1`. The fill operations produces a symbol with the same shape and type of the given first argument, but replacing all its components with the given second argument, `1` in this case. This is the **seed** needed to implement *reverse accumulation* algorithm for

automatic differentiation. This seed *forces* to compute the output produced by the original computation. It is possible to avoid this computation using the **optional** third argument of the `diff` function, given a seed symbol which will be treated as a new ground variable. When compiling or evaluation the differentiated symbols, you will need to give the value of the seed. The shape and type of the seed must be exactly the same as the non-differentiated symbol (in this case `d`), and normally filled with 1s, but it could be a seed produced as derivative by other function.

```
> seed  = AD.matrix('seed')
> dm,da = AD.diff(d, { m, a })
> = dm
(.* (+ (* 4 b) a) seed)
> = da
(.* m seed)
```

16.3 Symbol advanced methods

16.3.1 diff

```
table = s:diff(seed, table)
```

16.3.2 compile

```
s:compile(compiler)
```

16.3.3 arg_ipairs

```
Lua iterator = s:arg_ipairs()
```

This method returns a Lua iterator which traverses using `ipairs` all the arguments received by the given symbol. If it is a ground variable, this method returns an empty iterator.

16.3.4 replace

```
symbol = s:replace(new_symbol)
```

This method is used in optimization procedures. It allows to replace the caller symbol by the given symbol. It returns the caller symbol.

16.3.5 clear_var_name

```
symbol = s:clear_var_name()
```

This method removes the variable name associated with compilation. It returns the caller symbol.

16.3.6 set_dims

```
symbol = s:set_dims(d1, d2, ...)
```

This method indicates the shape of the caller symbol. It is useful with `matrix` type, allowing to check if matrices fit the declared operations. It returns the caller symbol.

```
> a,b = AD.matrix('a b')
> a:set_dims(2,4)
> b:set_dims(4,3)
> c = a * b
> = table.unpack( c.dims )
2    3
> a:set_dims(2,2)
> c = a * b
[string "luaopen_aprilann_autodiff"]:5: Incorrect matrix dims for multiplication: 2x2 * 4x3
stack traceback:
    [C]: in function 'assert'
    [string "luaopen_aprilann_autodiff"]:5: in function <[string "luaopen_aprilann_autodiff"]:5>
    (...tail calls...)
    stdin:1: in main chunk
    [C]: in ?
```

16.3.7 set_broadcast

```
symbol = s:set_broadcast(b1, b2, ...)
```

The term broadcasting describes how matrices with different shapes are treated during operations. Some mathematical operators allow to work with matrices with not fitted shapes, replicating one of the matrices over broadcasted dimensions. This concept is similar to numpy broadcast. As example, broadcasting is used in ANNs when you add bias vector, allowing to compute operations with several samples at the same time (mini-batch or bunch-mode):

```
> x,w,b = AD.matrix('x w b')
> x:set_dims(20, 4)  -- 20 features, 4 samples
> w:set_dims(10, 20) -- 10 outputs, 20 inputs
> b:set_dims(10, 1)  -- 10 bias components
> b:set_broadcast(false, true)
> output = w*x + b   -- bias is broadcasted to fit the addition operation
> = output:eval{ x=matrix(20,4):uniformf(),
                 w=matrix(10,20):uniformf(),
                 b=matrix(10,1):uniformf() }
 4.92572     6.4316      5.24994     5.73787
 5.13588     5.06926     4.93599     5.89694
 5.82963     6.1323      6.2065      7.06305
 4.51586     5.76797     5.27805     6.3306
 4.17794     5.31377     4.05488     4.4576
 4.20612     5.04744     4.77355     5.41917
 5.74113     6.56931     5.7724      6.52165
 6.00033     6.34304     5.77687     7.40072
 7.1957      8.11042     7.12002     8.2262
 5.53413     6.22189     5.72926     6.46156
# Matrix of size [10,4] in row_major [0x29c1750 data= 0x2979e00]
```

If you don't indicate `true` in the corresponding broadcasting dimension, an error will occur. By default, all the dimensions has a `false` broadcasting property.

```
> b:set_broadcast(false, false)
> output = w*x + b   -- bias is broadcasted to fit the addition operation
> = output:eval{ x=matrix(20,4):uniformf(),
```

```
                w=matrix(10,20):uniformf(),
                b=matrix(10,1):uniformf() }
[string "luaopen_aprilann_autodiff"]:5: Not broadcasted dimensions must be equal, found broadcasted_dims
stack traceback:
    [C]: in function 'assert'
    [string "luaopen_aprilann_autodiff"]:5: in function 'eval_func'
    [string "luaopen_aprilann_autodiff"]:1: in function <[string "luaopen_aprilann_autodiff"]:1>
    (...tail calls...)
    [C]: in ?
```

16.4 Symbols

16.4.1 Constants

16.4.2 Scalars

16.4.3 Matrices

16.4.4 Other data types

16.5 Gradient descent learning

Chapter 17

`autodiff.ann` package

17.1 Introduction

Related with package `require("aprilann.autodiff.ann")`.

Chapter 18

`matlab` package

18.1 Introduction

Package `matlab` could be loaded via the standalone binary, or in Lua with `require("aprilann.matlab)`.

The *MAT-file format* belongs to Matlab software. We follow this documentation to implement this loader. Saving is not available.

Currently, only **Cell Arrays**, **Structures**, and **Numeric Matrices** could be loaded, all of them only in **binary** format. Compression is allowed. All the data must follow the guidelines described at the documentation.

18.2 Test files

We use three test files (test1.mat, test2.mat, and test3.mat) produced by the following Matlab commands:

18.2.1 test 1

```
> x = [ -1.34187  -1.77726  -1.73478  ...
> -0.932328  0.59467  0.332692 ...
> ...
> ];
> save("test1.mat", "x")
```

18.2.2 test 2

```
> A = [ 1 2 3; 4 5 6 ];
> B = [ 7 8 9; 10 11 12 ];
> C = { A, B };
> save("test2.mat", "C")
```

18.2.3 test 3

```
> X.w = 1
> X.y = 2
```

```
> X.z = 3
> save("test3.mat", "X")
```

18.3 Basic operations

The *MAT-file* could be loaded using the function `matlab.read`. This function shows at the screen commented lines which indicates the kind of data loaded and the name of the variables. All the Matlab variables will be allocated at a Lua table, indexed by the name of the variable.

```
> a_table = matlab.read("test1.mat")
> print(a_table.x)
 1.34187    -1.77726    -1.73478    ...
-0.932328    0.59467     0.332692   ...
-0.254006   -2.86238     0.877438   ...
 ...         ...         ...        ...
```

It is also possible to print fields of the table using the `print` or `tostring` functions. The following example shows the `print` function for a *Cell Array*.

```
> a_table = matlab.read("test2.mat")
> print(a_table)
C : matlab.cell_array dims [1,2]
> print(a_table.C:get(1,1))
 1           2           3
 4           5           6
# MatrixDouble of size [2,3] stride [3,1] ref [0x22bebe0 data= 0x2206400]
> print(a_table.C:get(1,2))
 7           8           9
 10          11          12
# MatrixDouble of size [2,3] stride [3,1] ref [0x239f120 data= 0x2379ed0]
```

18.4 Loading matrices

When a *MAT-file* with MAT-matrix variables is loaded, every MAT-matrix is converted to *APRIL-ANN* matrix objects. Five matrices are available, depending on the MAT-matrix data-type: `matrix` for **float**, `matrixDouble` for **double**, `matrixInt32` for **int32**, `matrixComplex` for **float complex** numbers, and `matrixChar` for **char**.

18.5 Loading *Cell Arrays*

If any of the variables is a *Cell Array*, it becomes a Lua object (a table with methamethods) which has the following methods:

- `table = c:dim()` returns a table with the size of each dimension of the array.
- `number = c:dim(number)` returns the size of the given dimension (starging in 1)
- `element = c:get(p1,p2,...,pn)` returns the element at the position (p1,p2,...,pn), where element could be a `matrix`, `matrixChar`, `matrixInt32`, `cell_array`, or `structure`, depending on the class of data.

```
> a_table = matlab.read("test2.mat")
> print(a_table.C:get(1,1))
 1           2           3
 4           5           6
# Matrix of size [2,3] in row_major [0x27a0340 data= 0x26cf9c0]
> print(a_table.C:get(1,2))
 7           8           9
 10          11          12
# Matrix of size [2,3] in row_major [0x283c6a0 data= 0x26e4a40]
```

The following methods are for low-level access, which could be useful to do a loop over all the elements:

- `number = c:size()` returns the number of elements at the array.
- `element = c:raw_get(number)` returns the element at `row_major` sorted position `number`, being number between `0` and `c:size()-1`. The number is the position of the element if all elements where sorted as a continuous array.
- `table = c:compute_coords(number)` returns the coordinate position of a given raw position number. As previous method, the number must be between `0` and `c:size()-1`.

```
> a_table = matlab.read("test2.mat")
> print(a_table)
C : matlab.cell_array dims [1,2]
> C = a_table.C
> for i=0,C:size()-1 do e=C:raw_get(i)print("COORDS",unpack(C:compute_coords(i)))print(e)end
COORDS         1    1
 1           2           3
 4           5           6
# Matrix of size [2,3] in row_major [0x1904b20 data= 0x18c90f0]
COORDS         1    2
 7           8           9
 10          11          12
# Matrix of size [2,3] in row_major [0x19054e0 data= 0x18caed0]
```

18.6 Loading *Structures*

The *Structures* are transformed in Lua tables (as dictionaries), indexed by the name of the fields, and as values the corresponding elements. As before, the elements could be any kind of `matrix`, `cell_array`, or `structure`.

```
> a_table = matlab.read("test3.mat")
> print(a_table)
X : table
> print(a_table.X)
y : matrixDouble
z : matrixDouble
w : matrixDouble
> print(a_table.X.y)
 2
# Matrix of size [1,1] in row_major [0xd999c0 data= 0xd99690]
> print(a_table.X.w)
```

```
 1
# Matrix of size [1,1] in row_major [0xd99b60 data= 0xd99c20]
> print(a_table.X.z)
 3
# Matrix of size [1,1] in row_major [0xd99d30 data= 0xd99df0]
```

Chapter 19

`stats` package

19.1 Introduction

Package `stats` could be loaded via the standalone binary, or in Lua with `require("aprilann.stats)`. This package contains utilities for statistical purposes.

19.2 Matrix functions

19.2.1 stats.hist

19.2.2 stats.ihist

19.2.3 stats.amean

```
result = stats.amean(m[, dim])
```

19.2.4 stats.gmean

```
result = stats.gmean(m[, dim])
```

19.2.5 stats.hmean

```
result = stats.hmean(m[, dim])
```

19.2.6 stats.var

```
result = stats.var(m[, dim])
```

19.2.7 stats.std

```
result = stats.std(m[, dim])
```

19.2.8 stats.acf

```
result,lags = stats.acf(m[, { lag_max, lag_step, cor }])
```

19.2.9 stats.cov

```
result = stats.cov(x[, y[, { centered, true_mean }]])
```

19.2.10 stats.cor

```
result = stats.cor(x[, y[, { centered }]])
```

19.2.11 stats.percentile

```
r1,r2,...  = stats.percentile(m, p1, p2, ...)
```

19.2.12 stats.center

```
result,center = stats.center(m[, center])
```

19.2.13 stats.standardize

```
result,center,scale = stats.standardize(m[,{ center, scale }])
```

19.2.14 stats.summary

```
tbl = stats.summary(m)
```

19.3 Statistical distributions

19.3.1 stats.dist.uniform

19.3.2 stats.dist.normal

19.3.3 stats.dist.lognormal

19.3.4 stats.dist.binomial

19.3.5 stats.dist.bernoulli

19.3.6 stats.dist.beta

19.3.7 stats.dist.exponential

19.4 Running measures

19.4.1 Mean and variance class

stats.running.mean_var‘

This class is useful to compute mean and variance over a large number of elements in an efficient way, following (this method)[http://www.johndcook.com/standard_deviation.html] to avoid instability. This class has the following methods:

- `obj=stats.running.mean_var()` it is a constructor which builds an instance.
- `obj = obj:add(number)` a method for adds a number to the set. It returns the caller object.
- `obj = obj:add(iterator)` a method which adds the sequence of numbers returned by the given iterator function. It returns the caller object.
- `obj = obj:add(table)` a method which adds all the elements of the given table (as array) to the set. The elements could be numbers or functions. It returns the caller object.
- `mean,variance = obj:compute()` computes and returns the accumulated mean and variance from all the calls to `add` method.
- `number = obj:size()` returns the number of elements added.
- `obj = obj:clear()` re-initializes the object.

```
> obj = stats.running.mean_var()
> obj:add(4)
> obj:add(10)
> print(obj:compute())
7   18
> obj:add({2,8,6,24})
> print(obj:compute())
9   62
> obj:add( pairs({ a=2, b=10 }) )
> print(obj:compute())
8.25    50.785714285714
> print(obj:size())
8
```

19.5 Combinatorial

19.5.1 stats.comb

```
result = stats.comb(n,k)
```

19.5.2 stats.log_comb

```
result = stats.log_comb(n,k)
```

19.6 Bootstrapping and confidence intervals

19.6.1 boot

```
table = stats.boot{ ...  }
```

This function receives a data sample, performs a random resampling of the data and gives it to a statistic function. The procedure is repeated several times (bootstrapping technique), and the function returns a matrix with the statistics computed for all the repetitions.

The function receives a table with this fields:

- `size=number or table` the sample population size, or a table with several sample population sizes.
- `R=number` the number of repetitions of the procedure.
- `k=number` the number of results returned by `statistic` function. **By default** it is 1.
- `statistic=function` a function which receives a `matrixInt32` with a list of indices for resampling the data. The function can compute a number of k statistics (with k>=1), being returned as multiple results.
- `verbose=false` an **optional** boolean indicating if you want or not a verbose output. By default it is `false`.
- `seed=1234` an **optional** number indicating the initial seed for random numbers, by default it is `1234`.
- `random` an **optional** random number generator. Fields `seed` and `random` are forbidden together, only one can be indicated.
- `ncores=1` an **optional** number indicating the number of CPU cores to use for the computation. It allows to speed-up large bootstraps. By default it is `1`.

The following example takes `1000` random values and performs bootstrap resampling over them, computing the mean of the resampled population:

```
-- the random population
> errors = stats.dist.normal():sample(random(567),1000)
> -- executes the bootstrap resampling function
> resampled_data = stats.boot{
  size = errors:size(), R = 1000, seed = 1234, k=2,
  statistic = function(sample)
    local s = errors:index(1, sample)
    local var,mean = stats.var(s)
```

```
    -- this function returns two results, the mean and the variance (k=2)
    return mean,var
  end,
}
> -- the 95% confidence interval is computed, being the range [a,b]
> a,b = stats.boot.ci(resampled_data, 0.95, 1)
> print(a,b)
-0.073265952430665  0.051443199906498
```

19.6.2 boot.ci

`a,b = stats.boot.ci(data [, confidence=0.95 [, pos=1 ]])`

This function receives a matrix with data computed by `stats.boot` and returns the confidence interval for the given `confidence` value and the given statistic function position. It returns two numbers, the left limit, and the right limit, being the interval `[a,b]`.

```
-- the 95% confidence interval [a,b] of sample mean
local a,b = stats.boot.ci(resampled_data, 0.95, 1)
-- the 95% confidence interval [a,b] of sample variance
local a,b = stats.boot.ci(resampled_data, 0.95, 2)
```

19.7 Principal Components Analysis

19.7.1 pca.center_by_pattern

`matrix = stats.pca.center_by_pattern(matrix)`

This function modifies **in-place** the given `matrix`, centering all the patterns to have zero-mean. The patterns must be ordered by rows, and the features by columns. The function returns the same `matrix` as given.

```
> -- four patterns, 6 features
> m = matrix(4,6):uniformf(1,2)
> = m
 1.10795     1.55917     1.35379     1.22971     1.38055     1.81201
 1.1907      1.58711     1.38786     1.55067     1.01365     1.6242
 1.94076     1.33665     1.28377     1.72529     1.26619     1.60847
 1.64965     1.65704     1.55421     1.80517     1.68546     1.92166
# Matrix of size [4,6] in row_major [0x2575340 data= 0x249c490]
> stats.pca.center_by_pattern(m)
> = m
-0.299245    0.151974   -0.0534089  -0.177485   -0.0266466   0.404811
-0.201669    0.194743   -0.00450444  0.15831    -0.378713    0.231833
 0.413907   -0.190203   -0.243086    0.198439   -0.260668    0.081612
-0.0625433  -0.0551615  -0.15799     0.0929705  -0.0267389   0.209462
# Matrix of size [4,6] in row_major [0x2575340 data= 0x249c490]
> = m:sum(2):scal(1/m:dim(2)) -- each pattern is centered
-3.97364e-08
-1.58946e-07
-1.19209e-07
-1.39078e-07
# Matrix of size [4,1] in row_major [0x2519460 data= 0x2491bd0]
```

19.7.2 pca

```
U,S,VT = stats.pca(matrix)
```

This function implements standard PCA algorithm, based on covariance matrix computation, and Singular Values Decomposition (SVD). The function receives a `matrix`, where features are columns, and data samples are by rows, that is, for `M` patterns and `N` features, the `matrix` will be of `MxN` size. The function needs that input `matrix` was normalized, normally by centering the mean of each pattern (for example, using `stats.pca.center_by_pattern(...)` function, or other normalization depending in the task, see UFLDL Tutorial for more information about data normalization).

The function return three `matrix` objects, where:

- `U` is an orthogonal matrix which contains the eigen-vectors of the covariance matrix, with one eigen-vector per column, sorted in decreasing order.
- `S` is a vector (diagonal matrix) which corresponds with the singular values of the covariance matrix, sorted in decreasing order.
- `VT` is equivalent to `U`, and can be ignored.

```
> -- 10000 patterns, 100 features
> m = matrix(10000,100):uniformf(1, 2, random(1234))
> m = stats.pca.center_by_pattern(m)
> U,S,VT = stats.pca(m)
```

19.7.3 pca.threshold

```
i,s_i,s_mass = stats.pca.threshold(S [,mass=0.99] )
```

This function computes the cutting threshold for a given `S` vector with the singular values sorted in decreasing order. It receives the matrix, and an **optional** number with the probability mass which you want to retain, by default it is `mass=0.99`. The function computes three values:

- `i` is the index where you need to cut.
- `s_i` is the singular value at the index `i`, that is `S:get(i)`.
- `s_mass` is the mass accumulated until this index.

```
> = stats.pca.threshold(S, 0.5)
45  0.084392011165619   0.49575713463855
> = stats.pca.threshold(S, 0.7)
65  0.079482264816761   0.69391569135546
> = stats.pca.threshold(S)
97  0.06935129314661    0.98337004707581
```

19.7.4 pca.whitening

```
matrix = stats.pca.whitening(X, U, S [, epsilon=0.0] )
```

This function implments PCA whitening, given a data `matrix X` (patterns by rows, features by columns), the `U` and `S` `matrix` objects returned by `stats.pca(X)`, and an **optional** regularization value `epsilon` which by default is `epsilon=0.0`. The function returns a **new allocated** `matrix`, result of applying the whitening process.

```
> -- loading digits from TEST/digitos/digits.png
> img = ImageIO.read("TEST/digitos/digits.png")
> m = img:to_grayscale():matrix()
> ds = dataset.matrix(m, { patternSize={16,16}, stepSize={16,16},
                           numSteps={100,10} })
> data_matrix = ds:toMatrix():clone()
> data_matrix = stats.pca.center_by_pattern(data_matrix)
> -- PCA computation
> U,S,VT = stats.pca(data_matrix)
> -- PCA whitening
> data_whitened = stats.pca.whitening(data_matrix, U, S, 0.02)
> = data_whitened
Large matrix, not printed to display
# Matrix of size [1000,256] [0x1be71b0 data= 0x1b23540]
```

If you want, it is possible to do dimension reduction by given `U` and `S` matrix slices, instead of the whole matrices.

```
> -- PCA whitening and dimensionality reduction (256 => 100)
> out = stats.pca.whitening(data_matrix, U(':','1:100'), S('1:100'), 0.02)
> = out
Large matrix, not printed to display
# Matrix of size [1000,100] [0xe5c070 data= 0xe5c140]
```

See also documentation of `ann.components.pca_whitening` (when available).

19.7.5 zca.whitening

```
X = stats.zca.whitening(X,U,S,epsilon)
```

This function receives the same arguments as `stats.pca.whitening`, but instead of do a PCA whitening, it computes a ZCA whitening. In this case, the output is the given `matrix X`, so, the computation is done **in-place**.

```
> -- ZCA whitening
> data_whitened = stats.zca.whitening(data_matrix:clone(), U, S, 0.02)
> -- write to disk
> aux = data_whitened:clone("row_major")
> for sw in aux:sliding_window():iterate() do sw:adjust_range(0,1) end
> ImageIO.write(Image(aux), "digits-whitened.png")
```

19.7.6 pca.gs_pca

```
T,P,R = stats.pca.gs_pca{ ...  }
```

WARNING this implementation is in experimental stage.

Implementation of PCA-GS algorithm, an iterative efficient algorithm for PCA computation. This code is translated from GSL CBLAS implementation of the paper *Parallel GPU Implementation of Iterative PCA Algorithms, M. Andrecut*. The function receives a table with the following fields:

- `X=matrix`: a MxN `matrix`, M number of patterns, N pattern size.

- `K=number`: the number of components that you want to compute, K <= N.
- `max_iter=number`: the maximum number of iterations computing every component. It is an **optional** parameter, by default it is `max_iter=10000`.
- `epsilon=number` the convergence criterion. It is an **optional** parameter, by default it is `epsilon=1e-07`.

The function returns three matrices:

- The `T` scores matrix, with size MxK.
- The `P` loads matrix, with size NxK.
- The `R` residuals matrix, with size MxN.

19.8 confusion_matrix

`stats.confusion_matrix`

Chapter 20

stats.MI package

20.1 Introduction

Package `stats.MI` could be loaded via the standalone binary, or in Lua with `require("aprilann.stats.MI)`.

The table `stats.MI` contains functions useful to compute the Mutual Information between matrices of data. It has the following functions:

- `number = stats.MI.entropy(matrix, histogram, levels=256)` this function receives three optional arguments. The first two are related to the set of data from computing the entropy. One of them must be given, the other must be `nil`. The third argument is by default 256, and is only useful if the `matrix` is given, and indicates the number of levels for the histogram computation.
  - If the `matrix` argument is given, a histogram is computed to estimate the probability distribution of the data, using the given number of `levels`, 256 by default.
  - If the `histogram` argument is given, the function takes this histogram as the source for the probability distribution estimation.
- `MI,NMI = stats.MI.mutual_information(matrix1, matrix2, levels=256)` this function computes the amount of information mutually shared by the given two matrices of data, using `levels` for the histogram computation. The two matrices will be reinterpreted as a linear sequence of data, so the must have exactly the same size, and is recommended both matrices to being a vector of data, so multi-dimensional feature vectors are not allowed. The function returns the *Mutual Information*, and the *Normalized Mutual Information*.

```
> m1 = matrix(1,10):linear(1)
> m2 = matrix(1,10)
> m2:slice({1,1},{1,5}):linear(2)
> m2:slice({1,6},{1,5}):linear(2)
> print(m1)
 1          2          3          4          5          6          7          8          9          10
# Matrix of size [1,10] in row_major [0x260dae0 data= 0x260dbc0]
> print(m2)
 2          3          4          5          6          2          3          4          5          6
# Matrix of size [1,10] in row_major [0x260e280 data= 0x260de70]
> print(stats.MI.mutual_information(m1,m2))
2.321927794305  1.6989699041453
```

Chapter 21

`complex` package

21.1 Introduction

Package `complex` could be loaded via the standalone binary, or in Lua with `require("aprilann.complex)`.

The `complex` is a new kind of data added binded from C++, which could be used with `matrixComplex` and has available math operations in Lua and using CBLAS interface.

IMPORTANT as the `complex` data-type is a C++ object, it is available via a reference pointer, be careful because the assignation is done by reference, not by content.

21.2 Construction

Exists two possible constructors:

```
> c = complex(2,-1) -- 2 is real part, -1 is imaginary part
> print(c)
 2-1i
>
> -- the string is parsed in C++, worst performance than previous constructor
> c = complex("2+4i")
> print(c)
 2+4i
```

21.3 Math operations

The opreators '==', '*','/','+','-' *are defined to work with* `complex` *objects. If the other operand is a* number*, it is converted to a complex with only real part. If the other operand is a* string*, it will be converted to a complex number using the constructor from string.

Besides previous operations, the following math methods are available:

- `self = c:conj()` conjugates the given object. It is done **in-place**, so the object will be modified. Returns the caller object (self).
- `real,imaginary = c:plane()` returns the real and imaginary part.

- `number = c:real()` returns the real part of the number.
- `number = c:img()` returns the real part of the number.
- `abs,angle = c:polar()` returns the abs and angle of its polar form.
- `number = c:abs()` returns the 2-norm of the caller complex number.
- `number = c:angle()` returns the angle of its polar form.
- `other_complex = c:exp()` returns the exponential (e^z) of the caller complex number.
- `number = c:sqrt()` returns the square-root of the caller complex number.

```
> c1 = complex(1,2)
> c2 = complex(-4,-5)
> print(c1+c2)
 -3-3i
> print(c1*c2)
 6-13i
> print(c1-c2)
 5+7i
> print(c1:exp(), c2:exp())
 1.46869+2.47173i   -0.0119719+0.0175633i
> print(c1:abs(), c2:abs())
2.2360680103302 6.403124332428
```

21.4 Other methods

- `other = c:clone()` produces a new `complex` instance which has the same content as the caller.

Chapter 22

`util` package

22.1 Introduction

Package `util` could be loaded via the standalone binary, or in Lua with `require("aprilann.util)`. This package is the most important and dangerous. It extends standard Lua tables with new functionalities, and adds several utilities at GLOBALs table.

List of utilities added to Lua for scripting purposes.

22.2 Functions

Package `util` could be loaded via the standalone binary, or in Lua with `require("aprilann.util)`.

22.2.1 Serialization and deserialization of objects

22.2.1.1 util.serialize

```
str = util.serialize(obj)
util.serialize(obj, filename)
util.serialize(obj, stream)
```

22.2.1.2 util.deserialize

```
obj = util.deserialize(str)
obj = util.deserialize(filename)
obj = util.deserialize(stream)
```

22.2.1.3 util.to_lua_string

```
str = util.to_lua_string(obj, format)
```

22.2.2 Functional programming extensions

Lua ha been extended by the addition of new functions which works on the top of Lua iterators. Basic concepts as *map*, *reduce*, and *filter* has been implemented.

22.2.2.1 bind

```
func = bind(function, ...)
```

Allow to freeze any of the positional arguments of any function. The arguments of `bind` can be `nil`, and the returned function would merge its arguments with the list given to `bind` filling `nil` gaps adequately.

```
> f = bind(math.add, 2)
> = f(4)
6
> f = bind(math.div, nil, 3)
> = f(6)
2
```

22.2.2.2 reduce

```
whatever = reduce(function, initial_value, iterator)
```

The `reduce` function applies a `function` operator by pairs of values, the first argument is the accumulated value of the reduction until current iteration, and the second argument is value at current iteration. If the iterator returns two or more elements every time it is called, the second will be taken.

```
> value = reduce(math.min, math.huge, ipairs({4, 2, 1, 10}))
> print(value)
1
> value = reduce(function(acc,v) return acc*2+v end, 0, string.gmatch("01101", "." ))
> print(value)
13
```

22.2.2.3 apply

```
apply(func, iterator)
```

Applies a function to all the elements produced by the iterator. The function is called passing all the elements returned by one iterator call.

```
> t = { "a", "c", 3, 2 }
> apply(function(i,v1,v2) print(i,v1,v2) end, multiple_ipairs(t,t))
1   a       a
2   c       c
3   3       3
4   2       2
```

22.2.2.4 map

`table = map(func, iterator)`

Returns a table which is the result of apply the given function over all the items of the given iterator function.

```
> tmapped = map(bind(math.mul, 2), ipairs({1, 2, 3, 4}))
> print(table.concat(tmapped, " "))
2 4 6 8
```

22.2.2.5 map2

`table = map2(func, iterator)`

The same as the previous, but given the function the pair key,value.

```
> tmapped = map2(function(k,v) return k+v*2 end, ipairs({1, 2, 3, 4}))
> print(table.concat(tmapped, " "))
3 6 9 12
```

22.2.2.6 mapn

`table = mapn(func, iterator)`

The same as the previous, but given the function all the elements returned by the iterator at each iteration.

```
> tmapped = mapn(function(idx, ...) return table.pack(...) end,
>>          multiple_ipairs({1, 2, 3, 4},{5, 6, 7, 8}))
> for i,v in ipairs(tmapped) do print(i, table.concat(v, " ")) end
1   1 5
2   2 6
3   3 7
4   4 8
```

22.2.2.7 filter

`table = filter(func, iterator)`

Returns a table which contains only the elements produced by the iterator which were evaluated with true by the given `func` function. The function receives only one value.

```
> t = filter(function(v) return v%2 == 0 end, ipairs{1,2,3,4,5,6,7})
> print(table.concat(t, " "))
2 4 6
```

22.2.2.8 iterable_map

`another_iterator = iterable_map(func, iterator)`

Returns an iterator which every time is called maps the given function `func` using the given `iterator`. It allows multiple returned values from the given `iterator` (`map` and `map2` only allow pairs key,value).

Additionally, using `coroutine.yield(...)`, the mapping function could return more than one set of values at each iteration, allowing the implementation of ConcatMap iterators.

```
> -- standard map using iterable_map
> t = { Lemon = "sour", Cake = "nice", }
> for ingredient, modifier, taste in iterable_map(function(a, b)
>                                       return a:lower(),"slightly",b:upper()
>                                     end, pairs(t)) do
>   print(ingredient .." is ".. modifier .. " " .. taste)
> end
lemon is slightly SOUR
cake is slightly NICE
>
> -- ConcatMap iterator using iterable_map
> t = { Lemon = "sour", Cake = "nice", }
> for ingredient, modifier, taste in iterable_map(function(a, b)
>>                                       coroutine.yield(a:lower(),"very",b:upper())
>>                                       return a, "slightly", b
>>                                     end, pairs(t)) do
>>  print(ingredient .." is ".. modifier .. " " .. taste)
>> end
cake is very NICE
Cake is slightly nice
lemon is very SOUR
Lemon is slightly sour
```

The following example uses this function to extract all the words contained in a file:

```
> for str in iterable_map(function(line)
>>                           for _,str in ipairs(string.tokenize(line)) do
>>                             coroutine.yield(str)
>>                           end
>>                         end, io.lines("AUTHORS.txt")) do
>>  print(str)
>> end
In
this
project
has
been
worked:
-
Salvador
España
Boquera
-
Jorge
Gorbe
Moya
-
Adrián
Palacios
Corella
-
Joan
Pastor
Pellicer
```

```
-
Francisco
Zamora
Martínez
```

This function is taken from http://www.corsix.org/content/mapping-and-lua-iterators.

22.2.2.9 iterable_filter

`another_iterator = iterable_filter(func, iterator)`

Returns an iterator which every time is called filters using the given function `func` the elements produced by the given `iterator`. It allows multiple returned values from the given `iterator`.

```
> for v in iterable_filter(function(key,value) return value%2==0 end,
>>                          ipairs{1,2,3,4,5,6,7}) do
>>  print(v)
>> end
2
4
6
```

22.2.2.10 iterator class

The `iterator` class is developed to provide an easy and natural interface with previous and newer functions. The most important advantage is that `iterator` class relies always in Lua iterators, so, it is **lazy** in the way that the code is not executed *until* the iterator is *traversed*. `iterator` class is a wrapper of Lua iterators.

The following methods returns an iterator object or a Lua iterator:

- `obj = iterator(Lua iterator)`: the constructor receives an iterator, as for example the output of `ipairs` function, and returns an instance of `iterator` class.

```
> it = iterator(ipairs{ 1, 2, 3})
```

- `Lua iterator = obj:get()`: returns the current state of the underlying Lua iterator.
- `Lua iterator = obj()`: the same as previous method.

```
> it = iterator(ipairs{ 1, 2, 3})
> for k,v in it() do print(k,v) end
1   1
2   2
3   3
```

- `iterator = obj:map(func)`: this method is a wrapper of `iterable_map` function, and returns an instance of `iterator` class.

```
> it = iterator(ipairs{ 1, 2, 3}):map(function(k,v) return v*2 end)
> for v in it() do print(v) end
2
4
6
```

- `iterator = obj:filter(func)`: this method is a wrapper of `iterable_filter` function, and returns an instance of `iterator` class.

```
> it = iterator(range(1,50)):filter(function(n) return (n%10)==0 end)
> for v in it() do print(v) end
10
20
30
40
50
```

- `iterator = obj:field(...)`: this method receives a list of keys. It expects the underlying iterator to produce a list of tables. It returns an iterator which filters all the tables in the list taken the values at given keys, and returns a flatten list of values. There is an example below the following method.
- `iterator = obj:select(...)`: this method receives a list of numbers. It returns an iterator which selects only the output variables produced by the iterator at the given position numbers.

```
> layers = { { size=10 }, { size=100 } }
> iterator(ipairs(layers)):select(2):field("size"):apply(print)
10
100
```

- `iterator = obj:enumerate()`: enumerates the returned values, adding at first position a number.

```
> iterator(pairs{ a=4, b=3 }):enumerate():apply(print)
1   a   4
2   b   3
```

- `iterator = obj:iterate(func)`: this method is an specialization of `map` method for applying Lua iterator functions to each element of `obj`. The given `func` is expected to return an iterator over the given element. It is useful to do things like word counting:

```
> out =  iterator(io.lines("AUTHORS.txt")):
>> iterate(function(line) return string.gmatch(line, "[^\r\n\t ]+") end):
>> reduce(function(acc,w) acc[w] = (acc[w] or 0) + 1 return acc end,{})
> iterator(pairs(out)):apply(print)
has 1
Pastor  1
In  1
worked: 1
Palacios    1
-   5
España  1
Boquera 1
Joan    1
Francisco   1
Adrián  1
Martínez    1
been    1
Pellicer    1
Jorge   1
```

```
Zamora  1
Corella 1
this    1
Moya    1
Gorbe   1
Salvador    1
project 1
```

- `iterator = obj:call(funcname, ...)`: this method is a map over all the values by calling the method `funcname` (a string) using the given arguments. Because it is a method, the first argument of `funcname` will be each iterator value.

```
> for k in iterator(ipairs({ "h", "w" })):select(2):call("toupper"):get() do
    print(k)
  end
H
W
```

The following methods are **finalizers**, so, they return a value, not an iterator:

- `whatever = obj:reduce(func, initial_value)`: this method is a wrapper of `reduce` function.

```
> = iterator(range(1,50)):reduce(function(acc,a) return acc+a end, 0)
1275
> = iterator(range(1,50)):reduce(math.add, 0)
1275
```

- `obj:apply(func)`: this method is a wrapper of `apply` function.

```
> iterator(range(1,50)):filter(function(n) return (n%10)==0 end):apply(print)
10
20
30
40
50
```

- `string = obj:concat(sep1,sep2)`: concats all the elements using `sep1` and `sep2` strings. `sep1` is used to concat the elements of one iterator call. `sep2` is used to concat the elements between different iterations. By default, empty string will be used when `sep1` and `sep2` are `nil`. If only `sep1` is given, therefore `sep2=sep1`.

```
> = iterator(range(1,50)):filter(function(n) return (n%10)==0 end):concat(" ")
10 20 30 40 50
```

- `table = obj:table()`: returns a table with all the iterator values, using as key the first produced value, and the rest as value. If only one value is produced, the table will be indexed as an array.

```
> t = { "one", "two", "three" }
> p = iterator(ipairs(t)):map(function(k,v) return v,k end):table()
> iterator(pairs(p)):apply(print)
one     1
two     2
three   3
```

Using objects of this class, it is possible to produce code like this:

```
-- This example computes the dot product of two array tables. math.mul and math.sum are
-- auxiliary functions implemented in APRIL-ANN for the fast development of reductions.
> v = iterator(multiple_ipairs({1,2,3},{4,5,6})):select(2,3):
>>    map(math.mul):
>>    reduce(math.add, 0)
> print(v)
32
>
> -- The following code is equivalent without using iterator class
> v = reduce(function(a,b) return a+b end, 0,
>>           iterable_map(function(k,a,b) return a*b end,
>>                        multiple_ipairs({1,2,3},{4,5,6})))
> print(v)
32
```

22.2.3 Basic functions

22.2.3.1 april_list

`april_list(table)`

This function is this piece of code: `for i,v in pairs(table) do print(i,v) end`

22.2.3.2 april_help

`april_help(obj)`

Shows the documentation of the object given as argument. If the object is a class, you can access to instance methods by using `..` operator:

```
> -- using .. operator to access instance method get_state_table
> april_help(trainable.train_holdout_validation.."get_state_table")
method  Returns the state table of the training

description: Returns the state table of the training

outputs:
                 best Best trained model
           best_epoch Best epoch
       best_val_error Best epoch validation loss
        current_epoch Current epoch
                 last Last trained model
          train_error Train loss
     validation_error Validation loss

> -- showing help of the given class
> april_help(trainable.train_holdout_validation)
ID: trainable.train_holdout_validation
class  Training class using holdout validation

description: This training class defines a train_func which follows a training
```

```
            schedule based on validation error or in number of epochs. Method
            execute receives a function which trains one epoch and returns the
            trainer object, the training loss and the validation loss. This
            method returns true in case the training continues, or false if the
            stop criterion is true.
```

...

22.2.3.3 april_dir

`april_dir(string)`

This is a the same has `april_help`, but less verbose.

22.2.3.4 luatype

`luatype(whatever)`

The original `type` function is replaced by APRIL-ANN with a new function which returns the object id if it is a class instance. If you need to know the *exact* type given by Lua, this function is what you need.

22.2.3.5 check_version

`boolean = check_version(major,minor,commit)`

Checks if the version of the software is major.minor with the given commit, returning true if success, and returning false and showing a message in stderr otherwise.

22.2.3.6 april_print_script_header

`april_print_script_header(arg, file=stdout)`

This function writes at the given file (or stdout if not given) the given arg table (normally the arg received by the script), besides information about the HOST where the script is executed and the current DATETIME:

```
> april_print_script_header({ [0]="hello" })
# HOST:  django
# DATE:  dv jul  5 14:16:53 CEST 2013
# CMD:   hello
```

22.2.3.7 multiple_ipairs

`iterator,s,v = multiple_ipairs(...)`

Returns an iterator which traverses a several number of tables. If they don't have the same size, the remaining elements will be `nil`, ensuring that in all the iterations the number of returned elements is equals to the maximum size of given tables.

```
> for i,a,b,c in multiple_ipairs({1,2,3,4},{1,2},{3,4,5}) do print(i,a,b,c) end
1   1   1   3
2   2   2   4
3   3   nil 5
4   4   nil nil
```

22.2.3.8 multiple_unpack

```
...  = multiple_unpack( [table1, [table2, [...]]]  )
```

Allow to unpack multiple tables together, one at a time, and in a sequential fashion.

```
> print( multiple_unpack( {1,2,3}, {4,5}, {6,7,8,9} ) )
1   2   3   4   5   6   7   8   9
```

22.2.3.9 glob

```
table = glob(...)
```

Returns a list of filenames which match all the wildcard arguments received by the function.

```
> -- prints the name of all the files which have .lua or .h extensions
> for i,filename in ipairs(glob("*.lua", "*.h")) do print(filename) end
```

22.2.3.10 parallel_foreach

```
results = parallel_foreach(num_processes, iterator or array or number, func)
```

Executes a function over the given iterator (instance of class `iterator`), array table or the given number of repetitions, but forking the calling process in `num_processes`, improving the performance of the operation. **NOTE** that the parallelization is performed forking the caller process, so all child processes could access to the memory variables assigned and allocated before the fork, but they **don't share** the memory, it will be copied on write.

```
> t = map(function(v)return v end, 10)
> parallel_foreach(2, t, function(value) print(value*100) end)
200
400
600
800
1000
100
300
500
700
900
```

Additionally, if the function returns any value, this function would serialize the output of each process to a temporal file, and at the end, deserialize the content to the original process. This is useful when the overhead of serialization-deserialization procedure is less than the computing power needed by the processes.

```
> ret = parallel_foreach(2, 10, function(value) return value*100 end)
> print(table.concat(ret, "\n"))
100
200
300
400
500
600
```

```
700
800
900
1000
```

You can use iterators to control which data receives the called function:

```
> ret = parallel_foreach(2, iterator(ipairs{4,3,2,1}),
                         function(key,value) return key+value*100 end)
> print(table.concat(ret, "\n"))
401
302
203
104
>
> ret = parallel_foreach(2, iterator(ipairs{4,3,2,1}),
                         function(key,value) return key,key+value*100 end)
> print(table.tostring(ret))
{{1,401,["n"]=2},{2,302,["n"]=2},{3,203,["n"]=2},{4,104,["n"]=2}}
```

22.2.3.11 clrscr

`clrscr()`

Clears the screen.

22.2.3.12 printf

`printf(...)`

Equivalent to C `printf` function.

22.2.3.13 fprintf

`fprintf(file,...)`

Idem, but for the C `fprintf` function.

22.2.3.14 range

`range(inf,sup, step=1 )`

This function returns an iterator which starts at `inf`, ends at `sup`, and performs steps of the given `step` size.

```
> for i in range(10,20,2) do print(i) end
10
12
14
16
18
20
```

22.2.3.15 util.version

```
major,minor,commit = util.version()
```

Returns the version numbers.

22.2.3.16 util.omp_set_num_threads

```
util.omp_set_num_threads(number)
```

Modifies the number of threads for OMP.

```
> util.omp_set_num_threads(8)
```

22.2.3.17 util.omp_get_num_threads

```
number = util.omp_get_num_threads()
```

Returns the number of threads used by OMP.

```
> print(util.omp_get_num_threads())
8
```

22.2.4 Math table extensions

22.2.4.1 math.add

```
number = math.add( a ,b )
```

Returns the result of `a+b`.

```
> = math.add(2,3)
5
```

22.2.4.2 math.sub

```
number = math.sub( a, b )
```

Returns the result of `a-b`.

```
> = math.sub(2,3)
-1
```

22.2.4.3 math.mul

```
number = math.mul( a, b )
```

Returns the result of `a*b`.

```
> = math.mul(2,3)
6
```

22.2.4.4 math.div

`number = math.div( a, b )`

Returns the result of `a/b`.

```
> = math.div(2,3)
0.66666666666667
```

22.2.4.5 math.eq

`number = math.eq( a, b )`

Returns the result of `a==b`.

22.2.4.6 math.lt

`number = math.lt( a, b )`

Returns the result of `a<b`.

22.2.4.7 math.le

`number = math.le( a, b )`

Returns the result of `a<=b`.

22.2.4.8 math.gt

`number = math.gt( a, b )`

Returns the result of `a>b`.

22.2.4.9 math.ge

`number = math.ge( a, b )`

Returns the result of `a>=b`.

22.2.4.10 math.land

`number = math.land( a, b )`

Returns the result of `a and b`.

22.2.4.11 math.lor

`number = math.lor( a, b )`

Returns the result of `a or b`.

22.2.4.12 math.lnot

```
number = math.lnot( a )
```

Returns the result of `not a`.

22.2.4.13 math.round

```
number = math.round(number)
```

Returns the rounding integer number for the given real number.

```
> = math.round(4/3)
1
```

22.2.4.14 math.clamp

```
number = math.clamp(value,lower,upper)
```

Clamp the given value to be between [lower,upper], if it is out of the range, it is forced to be at the limit.

```
> print(math.clamp(15,3,6), math.clamp(0,3,6), math.clamp(4,3,6))
6       3       4
```

22.2.5 String table extensions

22.2.5.1 Operator %

Inspired in penlight library, a Python-like operator `%` has been defined. It allows to produce formatted strings, and implements map-like substitutions:

```
> = "$obj1 = %.4f\n$obj2 = %.4f" % {20.4, 12.36, obj1="cat", obj2="dog"}
cat = 20.4000
dog = 12.3600
```

22.2.5.2 truncate

```
string = str:truncate(columns, prefix)
```

22.2.5.3 basename

```
string = path:basename()
```

Returns the basename (the last filename) of a given path.

```
> print(("/a/path/to/my/file.txt"):basename())
file.txt
```

22.2.5.4 dirname

Returns the path, removing the basename.

```
> print(("/a/path/to/my/file.txt"):dirname())
/a/path/to/my/
```

22.2.5.5 remove_extension

`string,string = path:remove_extension()`

Removes the extension of the filename in the given path, and returns the path without the extension and the extension string.

```
> print(("/a/path/to/my/file.txt"):remove_extension())
/a/path/to/my/file  txt
```

22.2.5.6 get_extension

`string = path:get_extension()`

Returns only the extension of the given path string.

```
> print(("/a/path/to/my/file.txt"):get_extension())
txt
```

22.2.5.7 get_path

`string = path_with_filename:get_path(sep)`

Synonim of `dirname()`.

22.2.5.8 lines_of

`string = str:lines_of()`

Returns an iterator function which traverses the given string splited by newline character.

```
> for line in ("one\ntwo"):lines_of() do print(line) end
one
two
```

22.2.5.9 chars_of

`iterator = str:chars_of()`

Returns an iterator function which traverses the given string splited by chars.

```
> for i,ch in ("one two"):chars_of() do print(i,ch) end
1   o
2   n
3   e
4
5   t
6   w
7   o
```

22.2.5.10 tokenize

```
table = str:tokenize(sep=' \t\n\r')
```

Returns a table with the string tokenized using the given `sep` set of characters.

```
> for i,token in ipairs((" one\ntwo\tthree four"):tokenize("\t\n ")) do print(i,token) end
1   one
2   two
3   three
4   four
> for i,token in ipairs(string.tokenize(" one\ntwo\tthree four", "\n ")) do print(i,token) end
1   one
2   two three
3   four
```

22.2.5.11 tokenize_width

```
table = str:tokenize_width(width=1)
```

22.2.5.12 join

The `string.join` function is equivalent to Lua `table.concat` function.

22.2.6 Table table extensions

22.2.6.1 table.insert

```
table = table.insert(table,value)
```

The original `table.insert` function was replaced with a new one which returns the table given as first argument. It is combinable with `reduce` function.

22.2.6.2 table.luainsert

```
table.luainsert(table,value)
```

The original Lua `table.insert` function.

22.2.6.3 table.clear

`table.clear(table)`

Removes all the elements of a table, but it doesn't forces Lua to deallocate the memory. This function is useful if you want to reuse a table variable several times inside a loop, it is better to clear the table than to allocate a new one table.

```
> t = {}
> for i=1,1000 do table.clear(t) STUFF USING t end
```

22.2.6.4 table.unpack_on

`table.unpack_on(table, dest_table)`

This function puts the fields of the given table at the table `dest_table`. It is useful to put table fields on the global scope of Lua.

```
> print(a, b, c)
nil nil nil
> t = { a=1, b=2, c=3 }
> table.unpack_on(t, _G)
> print(a, b, c)
1   2   3
```

22.2.6.5 table.invert

`table = table.invert(table)`

Returns the table resulting from the inversion of key,value pairs of the given table argument.

```
> t = { "a", "b", "c" }
> t_inv = table.invert(t)
> for i,v in pairs(t_inv) do print(i,v) end
a   1
c   3
b   2
```

22.2.6.6 table.slice

`table = table.slice(t, ini, fin)`

Returns from the given table the slice of elements starting at `ini` and finishing at `fin`.

```
> t = { 1, 2, 3, 4, 5, 6, 7, 8, 9 }
> print(unpack(table.slice(t, 2, 4)))
2   3   4
```

22.2.6.7 table.search_key_from_value

```
key = table.search_key_from_value(table,value)
```

This function searchs a value at the given table and returns its key. If the value is repeated (obviously using different keys), any of the possible keys will be returned, but it is not possible to determine which one.

```
> print(table.search_key_from_value({ a=15, b=12 }, 12))
b
```

22.2.6.8 table.reduce

```
whatever = table.reduce(table,function,initial_value)
```

Equivalent to `reduce(function, initial_value, pairs(table))`.

22.2.6.9 table.imap

```
table = table.imap(table,function)
```

Equivalent to `map(function, ipairs(table))`.

22.2.6.10 table.map

```
table = table.map(table,function)
```

Equivalent to `map(function, pairs(table))`.

22.2.6.11 table.imap2

```
table = table.imap2(table,function)
```

Equivalent to `map2(function, ipairs(table))`.

22.2.6.12 table.map2

```
table = function table.map2(table,function)
```

Equivalent to `map2(function, pairs(table))`.

22.2.6.13 table.ifilter

```
table = table.ifilter(table,function)
```

This functions traverses the given table as an array (using ipairs function), and returns a new table which contains only the elements where the given function returns true. The function is called passing the pair key,value as two arguments.

22.2.6.14 table.filter

```
table = table.filter(table,function)
```

Idem as the previous one but for general tables (using pairs functions).

22.2.6.15 table.join

```
table = table.join(t1,t2)
```

Returns a table which is the concatenation of the two given tables.

```
> t = table.join({1,2,3}, {10,11,12})
> print(table.concat(t, " "))
1 2 3 10 11 12
```

22.2.6.16 table.deep_copy

```
table = table.deep_copy(table)
```

Returns a table which is a deep copy of the Lua data-values contained at the given table, and a shallow copy (copied by reference) of its C++ references.

22.2.6.17 table.linearize

```
table = table.linearize(table)
```

Converts an unsorted dictionary in an array, throwing away the keys. The order of the array is not determined.

22.2.6.18 table.tostring

```
string = table.tostring(table)
```

This function converts the given table to a string which contains the table values, and which could be loaded as a Lua chunk. It only works with tables which doesn't contain C++ references.

```
> t = { 1, 2, a={ ["foo"] = "bar" } }
>  print(table.tostring(t))
{
[1]=1,[2]=2,["a"]=
{
["foo"]="bar"
}
}
```

22.2.6.19 table.max

```
number,index = table.max(table)
```

This function returns the maximum value and the index of the key which contains it. The table is traversed using pairs function.

22.2.6.20 table.min

```
number,index = table.min(table)
```

This function returns the minimum value and the index of the key which contains it. The table is traversed using pairs function.

22.2.6.21 table.argmax

`index = table.argmax(table)`

This function is equivalent to `table.max` returning only the index.

22.2.6.22 table.argmin

`index = table.argmin(table)`

This function is equivalent to `table.min` returning only the index.

22.2.7 Io table extensions

22.2.7.1 io.uncommented_lines

`iterator = io.uncommented_lines( [filename] )`

Returns a function iterator which traverses the given filename (if not given, it uses `io.stdin`), removing the lines which begins with `#` symbol.

```
> for line io.uncommented_lines() do STUFF end
```

22.3 Miscellaneous classes

22.3.1 util.stopwatch

`util.stopwatch`

22.3.2 util.vector_uint

`util.vector_uint`

22.3.3 util.vector_float

`util.vector_float`

Chapter 23

gzio package

23.1 Introduction

Package gzio could be loaded via the standalone binary, or in Lua with require("aprilann.gzio).

23.2 gzio class, GZip files

NOTE that io.open is overwritten by APRIL-ANN to automatically open gzipped files by using gzio class, if the filename has .gz extension.

The gzio class is compatible with standard Lua file. See Lua documentation for more details

- obj = gzio.open(path,mode="r") constructs the object and opens the given path using the given mode.
- obj = io.open(path,mode="r") opens the given path using the given mode, and returns a gzio object if the file has .gz extension, otherwise it returns a Lua file.
- obj:close() closes the file.
- obj:flush() flushes the file.
- position = obj:seek(whence="cur",offset=0) moves the cursor from the given base position whence plus the given offset. The whence could be "cur" or "set", the "end" value is forbidden in ZLib. It returns the position of the cursor at the file.
- value,... = obj:read(format="*l", ...) reads a sequence of values from the file, following the given format strings.
  - "*l" reads a line.
  - "*n" reads a number.
  - "*a" reads the whole file.
  - NUMBER reads a string with a maximum of NUMBER bytes.
- obj:write(value, ...) write the given sequence of values to the file. A valid value is a string or a number.
- iterator = obj:lines(...) returns an iterator which read by lines following the given values, by default "*l". The file is not closed at end.

- `iterator = io.lines(path, ..)` returns an iterator which traverse the given path by lines, following the given values, by default "*l". Read Lua documentation for details. This function uses `gzio` object if the file has `.gz` extension, otherwise it uses the standard `io.lines`.

23.3 `tar` class, TAR files

Chapter 24

Image package

24.1 Introduction

This package is available with `require("aprilann.Image")`.

Two globals are declared in this package: `Image` and `ImageRGB`. Both classes represent its internal data using floats, in the case of `Image` with one float, in the case of `ImageRGB` with 3 floats (Red, Green, Blue). So, usually 8 bits per color images will be transformed to be the color `0=0.0` and the color `255=1.0`.

24.2 Serialization and deserialization

Images **cannot** be serialized as a Lua object into a `.lua` filename using `util.serialize()` and `util.deserialize()` functions. Instead of that, it is possible to write images into `png`, `tiff` or `jpeg` files by using the package `ImageIO`. This package allows to write `Image` or `ImageRGB` images.

24.3 Visualization

A tool module has been deployed to facilitate visualization of images and matrices. It depends on **lgi** binding library for Lua 5.2, and uses the GTK binding. In order to do the `require "tools.lgi.gtk"` you need to have the APRIL-ANN *ROOT* directory in your `package.path` Lua variable, or in your `LUA_PATH` environment variable. By default, the current directory is included, so if you execute the `require` being your working directory the APRIL-ANN GIT root directory, it will work. In any case, **NEVER** add the april_tools directory to the LUA_PATH, because the *lgi* module will collide with the *tools.lgi* module.

```
> gtk = require "tools.lgi.gtk"
> gtk.show("sheared.png")
```

If it doesn't work, try add the APRIL-ANN ROOT directory to the `package.path` variable.

```
> package.path = "YOUR_APRIL_ANN_PATH/?.lua;" .. package.path
> gtk = require "tools.lgi.gtk"
> gtk.show("sheared.png")
```

24.3.1 gtk.show(img1, img2, ...)

This function allows to visualize images using GTK library. It receives a variable number of arguments, and every argument will be displayed in a different window. Every argument could be:

- A filename, opening the contained image.
- A `matrix`, which needs to be bi-dimensional. It will be converted to an `Image` and then showed.
- An `Image` or `ImageRGB`.

Note that the Lua command shell will be blocked after executing GTK main loop. Any of the created windows has an exit button to allow closing the GTK main loop.

24.4 Image class

The `Image` class allows to work with gray-scale images. Every image contains an underlying `matrix` instance (floats).

24.4.1 Constructor: img = Image(matrix)

The constructor receives a `matrix` where the image is contained. This matrix must be **simple**, that is, contiguous in memory, and in `row_major`. Given a matrix with data, it is possible to load an `Image` with:

```
> m = matrix(100,100)
> -- make a gradient
> for sw in m:sliding_window():iterate() do sw:linear():adjust_range(0,1) end
> -- constructor for Image
> img = Image(m)
> = img
# Image 100x100+0+0 [0x2490810 data= 0x25843e0]
> -- you can see the image writing it with ImageIO or using the gtk module
> ImageIO.write(img, "gradient.png")
```

24.4.2 Image = img:crop(string)

This method allows to select a portion of an `Image`, and returns a new instance which **references** the given portion. The crop is given by using a string (like in *convert* command), with the following format: `<width>x<height>{+-}<x>{+-}<y>`.

```
> img2 = img:crop("20x20+10+10")
> = img2
# Image 20x20+10+10 [0x2490810 data= 0x25843e0]
```

24.4.3 Image = img:crop(width, height, offsetx, offsety)

Similar to previous one, but given the crop using four arguments.

```
> img2 = img:crop(20, 20, 10, 10)
> = img2
# Image 20x20+10+10 [0x2490810 data= 0x25843e0]
```

24.4.4 matrix = img:matrix()

This method returns the underlying `matrix`. **Be careful** with this method, the memory is shared between the `Image` object and the returned `matrix` object.

```
> m2 = img:matrix()
> = m2
...
# Matrix of size [100,100] in row_major [0x2490810 data= 0x25843e0]
> = m -- the matrix references the original
...
# Matrix of size [100,100] in row_major [0x2490810 data= 0x25843e0]
```

24.4.5 width,height,offsetx,offsety = img:geometry()

This method returns the geometry information of the image.

```
> = img2:geometry()
20  20  10  10
```

24.4.6 number = img:getpixel(x,y)

This method returns the given position (x,y) pixel value.

24.4.7 img:putpixel(x,y,number)

This method assigns the given value at the given (x,y) pixel position.

24.4.8 Image = img:clone()

This method returns a **deep-copy** of the caller object.

24.4.9 matrix = img:projection_h()

This method returns the horizontal projection of the caller object.

24.4.10 matrix = img:projection_v()

This method returns the vertical projection of the caller object.

24.4.11 Image = img:shear_h(angle, [, units="rad" [, WHITE=0.0]])

This method returns an `Image` which is a shear transformation with the given `angle`. Optionally, a **second argument** could be given indicating with the string `rad`, `deg` or `grad` the angle unit, by default it is `rad`. The **thrird argument** is also optional, and it indicates which pixel value is taken as white color, by default is `0.0`.

```
> img_sh = img:shear_h(0.1)
> ImageIO.write(img_sh, "sheared.png")
```

24.4.12 img:shear_h_inplace(angle, [, units="rad" [, WHITE=0.0]])

This method applies the same transformation as the `img:shear_h(...)`, but instead of returning a new `Image`, the transformation is performed **in-place**.

24.4.13 w,h,x,y = img:min_bounding_box(threshold)

This method returns the bounding box of the caller object, using `threshold` for deciding when a value is considered as background or not.

24.4.14 img:copy(Image, dest_x, dest_y)

This method copies the given `Image` in the given (dest_x,dest_y) position of the caller object.

24.4.15 Image = img:substract(Image)

This method applies image substraction.

24.4.16 img:threshold(low, high)

This method transforms **in-place** the caller object with the given range for black/white thresholding.

24.4.17 Image = img:rotate90cw(param)

This method returns a new `Image` which is a rotation of 90^o in clock-wise direction (if `param=1`) or in counter-clock-wise (if `param=-1`) of the caller object.

24.4.18 Image = img:invert_colors()

This method returns an image with the colors inverted.

24.4.19 Image = img:remove_blank_columns()

24.4.20 Image = img:add_rows(top, bottom, value)

24.4.21 Image = img:convolution5x5(table [, default_value])

24.4.22 Image = img:resize(x,y)

24.4.23 Image,x,y = img:affine_transform(AffineTransform2D, default_value)

24.4.24 ImageRGB = img:to_RGB()

24.4.25 matrix = img:comb_lineal_forward(x, y, w, h, w2, h2, LinearCombConFloat)

24.5 ImageRGB class

This class is an instantiation of the same C++ template used for `Image` class. Both classes has similar methods.

24.5.1 Constructor: ImageRGB = ImageRGB(matrix)

`ImageRGB` class could be loaded from a `matrix` with 3 dimensions. The last dimension of the `matrix` has size 3 for the componentes Red, Green, Blue. **Be careful** with this constructor, the memory is shared between the `ImageRGB` object and the given `matrix` object.

```
> img_rgb = ImageRGB(matrix(100,100,3):linear())
> = img_rgb
# ImageRGB 100x100+0+0 [0x2535700 data= 0x251fc40]
```

24.5.2 Methods shared with Image class

The following methods are shared between both classes, and their has the same interface:

- `ImageRGB = img_rgb:crop(string)`
- `ImageRGB = img_rgb:crop(width, height, offsetx, offsety)`
- `width,height,offsetx,offsety = img_rgb:geometry()`
- `r,g,b = img_rgb:getpixel(x,y)`
- `img_rgb:putpixel(x,y, r,g,b)`
- `ImageRGB = img_rgb:clone()`
- `ImageRGB = img_rgb:shear_h(angle, [, units="rad" [, WHITE=0.0 ] ] )`
- `img_rgb:shear_h_inplace(angle, [, units="rad" [, WHITE=0.0 ] ] )`
- `img_rgb:copy(ImageRGB, dest_x, dest_y)`
- `ImageRGB = img_rgb:rotate90cw(param)`

- `ImageRGB = img_rgb:invert_colors()`
- `ImageRGB = img_rgb:convolution5x5(table [, r,g,b ])`
- `ImageRGB = img_rgb:resize(x,y)`
- `ImageRGB,x,y = img_rgb:affine_transform(AffineTransform2D, r,g,b)`

24.5.3 `matrix = img_rgb:matrix()`

This method returns the underlying `matrix`. **Be careful** with this method, the memory is shared between the `ImageRGB` object and the returned `matrix` object.

24.5.4 `Image = img_rgb:to_grayscale()`

This method returns an instance of `Image`, transforming the 3 color components into a gray component.

24.6 Useful examples

A usual situation is to load a `matrix` from a PNG image (in RGB color, with 0=BLACK, and 1=WHITE), but transform it to train ANNs in gray-scale with 1=BLACK and 0=WHITE. This could be done by the following code:

```
> img_matrix = ImageIO.read(filename):to_grayscale():invert_colors():matrix()
```

Chapter 25

ImageIO package

25.1 Introduction

This package is avaialable as `require("aprilann.ImageIO")`.

ImageIO package implement functions for read/write of images in any supported format. Currently PNG and TIFF formats are supported.

25.2 Functions

25.2.1 ImageRGB = ImageIO.read(filename [, img_format])

This function allows to read an image from a given `filename`. It returns an instance of `Image` class. The `img_format` is **optional**, if not given, it will be extracted from the extension of the given `filename`. If given, it must be a string like `png`, `tiff` or `tif`. More formats will be supported in the future.

25.2.2 ImageIO.write(Image|ImageRGB, filename [, img_format])

This function allows to write an image into a given `filename`. The `img_format` is **optional**, if not given, it will be extracted from the extension of the given `filename`. If given, it must be a string like `png`, `tiff` or `tif`. More formats will be supported in the future.

Chapter 26

AffineTransform2D package

26.1 Introduction

This package is avaialable as `require("aprilann.AffineTransform2D")`.

26.2 AffineTransform2D class

26.2.1 Constructor

```
> AffineTransform2D()
> AffineTransform2D(matrix)
```

26.2.2 obj = obj:accumulate(other)

26.2.3 obj = obj:rotate(angle [, center_x, center_y])

26.2.4 obj = obj:scale(sx, sy)

26.2.5 obj = obj:translate(x, y)

26.2.6 obj = obj:shear(angle_x, angle_y)

26.2.7 dstx,dsty = obj:transform(x, y)

Chapter 27

`class` package

27.1 Introduction

Package `class` is a copy of the code avaiable at Lua OOP-iter for deploying OOP to Lua. It could be loaded by using `require "aprilann.class"`, or if you have installed the Lua OOP-iter module, by doing `class = require "oop-iter.class"`.

See test-class.lua file for an example of how to use this package. More updated version of this documentation is in Lua OOP-iter README.md, at `class` module section.

27.2 Description

The class module implements OOP for Lua in a similar way as luabind does for C++ classes. So, the class module functions are compatible with it. The OOP is implemented by defining several tables for each desired class. Every class has a name, which allow to store it into a weak table in order to retrieve the class by its name in any moment. A class is defined by doing:

```
class_table, methods_table = class('myClassName'[, parent_class_table[, class_table]])
```

Two tables are returned as result of this call, the `class_table` which allows to construct instances by using the `class_table` as a function (it has implemented the `__call` metamethod), and a `methods_table` where instance methods must be defined. Class methods will be defined into `class_table`, and special names constructor/destructor allow to define the behavior of the object at these specific stages. So, the first is to define a constructor and a destructor (NOTE: both are optional):

```
class_table:constructor(whatever) self.blah = whatever end
class_table:destructor() free_resource(self.blah) end
```

In the same way, instance methods will be defined in `methods_table`:

```
methods_table:my_method() return self.blah end
```

Additionally, instance metamethods can be defined using the `class.extend_metamethod` function. Be carefule, `__gc` and `__index` metamethods are defined by default and them cannot be modified, any change will produce an unexpected behavior:

```
class.extend_metamethod(class_table, "__tostring", function() print("foo") end)
```

Looking with more detail inside the architecture, the `class(...)` function call defines the following hierarchy of tables:

```
  class_table = {
    constructor = default_constructor, -- it does nothing
    destructor = default_destructor,   -- it does nothing
    -- the meta_instance table contains the metatable of instance objects
    meta_instance = {
      id  = class_name_string,
      cls = class_table_reference,
      __tostring = default_tostring_metamethod, -- it is safe to be overwritten
      __index = methods_table, -- the table where instance methods are defined
      __gc = default_gc_metamethod,
    }
  }

  -- class_table metatable contains:
  {
    id     = class_name .. " class",
    parent = parentclass, -- if given any
    __tostring = default_tostring_metamethod,
    __concat   = default_concat_metamethod,
    __call     = constructor_call,
  }
```

The `class(...)` function call returns first the `class_table` and second the `class_table.meta_instance.__index` field, letting the user to define class methods and instance methods there. By default constructor and destructor functions does nothing, and they are implemented at `class_table.constructor` and `class_table.destructor` fields. `class_table.meta_instance` table can be safety modified by calling to `class.extend_metamethod(...)`, or writing non-safety manual changes into `class_table.meta_instance`.

27.2.1 Simple inheritance

Simple inheritance has been implemented by defining a metatable for the `class_table.meta_instance.__index` table. Having a class table `myClass1`, you can define the class `myClass2` as a child of previous one by writing:

```
> -- parent class
> myClass1,myClass1Methods = class("myClass1")
> myClass1:constructor(...) whatever stuff here end
> -- derived or child class
> myClass2,myClass2Methods = class("myClass2", myClass1)
> myClass2:constructor(...) myClass1.constructor(self, ...) more stuff here end
```

Note that parent constructor call is not made by default, and `myClass2:constructor` calls explicitly `myClass1.constructor` function passing the `self` reference. In this way, whatever construction stuff done in `myClass1` will be done for `myClass2`. It is not mandatory to do this, but in many cases it will be helpful. However, you can build `myClass2` instances in whatever way you want if the result is compatible with the methods inherited from `myClass1`.

`myClass2Methods` can overwrite or not methods defined at `myClass1Methods`. Non overwritten methods will be delegated calling `myClass1` implementation, so be careful to ensure both objects are compatible.

Destructors are called following the hierarchy, first child destructor and after the parent class.

27.3 Reference

The following public functions are available:

27.3.1 object = class(name[, parent_class[, class_table]])

Creates a class table with a given class_name. It receives an optional parent class to implement simple inheritance. It returns the class table; another table which will contain the methods of the object. Constructor and destructor methods will be declared into the class table as `class_name:constructor(...)` and `class_name:destructor()`. Additionally, a third optional argument is given, which allows to give a predefined `class_table`, useful is you want to make global instead of local variables, or to convert into a class an existent table.

```
> -- a simple class with name cls1
> cls1,cls1_methods = class("cls1")
> -- a derived class from cls1
> cls2,cls2_methods = class("cls2")
> -- a nested class defined into cls2 table
> cls2.nested1 = {}
> nested1,nested1_methods = class("cls2.nested1", nil, cls2.nested1)
> -- a derived nested class
> cls2.nested2 = {}
> nested2,nested2_methods = class("cls2.nested2", cls2.nested1, cls2.nested2)
```

A class_name cannot be used two times, that is, a class can't be redefined. If you need to redefine a class, use `class.forget(class_name)` before. Otherwise the following error message will be displayed:

```
> class("cls1")
> class("cls1")
./oop-iter/class.lua:40: cls1 class name exists
stack traceback:
    [C]: in function 'assert'
    ./oop-iter/class.lua:40: in function 'register_class_table'
    ./oop-iter/class.lua:289: in function 'class'
    stdin:1: in main chunk
    [C]: in ?
```

27.3.2 boolean = class.is_a(object, class_table)

Predicate which returns true if a given object instance is a subclass of a given Lua class table.

```
> cls1,cls1_methods = class("cls1")
> cls2,cls2_methods = class("cls2")
> cls3,cls3_methods = class("cls3", cls1)
> cls3 = cls3()
> = class.is_a(o1, cls1)
true
> = class.is_a(o1, cls2)
false
> = class.is_a(o1, cls3)
true
```

27.3.3 super_class_table = class.super(class_table)

Returns the super class table of a given derived class table. Throws an error if the given class has not a super class.

```
> cls1,cls1_methods = class("cls1")
> cls2,cls2_methods = class("cls2", cls1)
> = ( class.super(cls2) == cls1 )
true
```

27.3.4 class_table = class.of(object)

Returns the class table of the given object instance. In case the given parameter is a Lua value but not an object, it returns `nil`. So, this method can be used also to ask if a Lua value is or not an object.

```
> cls1,cls1_methods = class("cls1")
> o = cls1()
> = ( class.of(o) == cls1 )
true
> = class.of( {} )
nil
> = class.of( 5 )
nil
```

27.3.5 class.extend(class_table, key, value)

Extends the given class table with the addition of a new key=value pair into the object instance table. It throws an error if the 1st parameter is not a class table.

```
> cls1,cls1_methods = class("cls1")
> foo = function() end
> class.extend(cls1, "foo", foo)
> ( cls1_methods.foo == foo )
true
```

27.3.6 class.extend_metamethod(class_table, key, value)

Extends the given class table with the addition of a new key=value pair into the object `meta_instance` table, where metamethods are stored. It throws an error if the 1st parameter is not a class table. Be careful, several metamethods (`__index`, `__gc`) and keys (`id`, `cls`) are defined by default in order to implement OOP, overwritten them will produce unexpected errors. The call will throw an error if you try to overwrite any of them. However, `__tostring` metamethod is also defined but it is totally safe to overwrite it.

```
> cls1,cls1_methods = class("cls1")
> foo = function() return "Hello world!" end
> class.extend_metamethod(cls1, "__concat", foo)
> o = cls1()
> = o .. o
Hello world!
```

27.3.7 value = class.consult(class_table, key)

Returns the value associated with the given key at the given class_table. Throws an error if the 1st parameter is not a class table.

```
> cls1,cls1_methods = class("cls1")
> cls1_methods.foo = function() end
> = ( class.consult(cls1, "foo") == cls1_methods.foo )
true
```

27.3.8 value = class_table .. key

Equivalent to previous one.

```
> cls1,cls1_methods = class("cls1")
> cls1_methods.foo = function() end
> = ( cls1.."foo" == cls1_methods.foo )
true
```

27.3.9 value = class.consult_metamethod(class_table, key)

Returns the value associated with the given key at the given class_table meta_instance (instance metatable). Throws an error if the 1st parameter is not a class table.

```
> cls1,cls1_methods = class("cls1")
> foo = function() return "Hello world!" end
> class.extend_metamethod(cls1, "__concat", foo)
> = ( class.consult_metamethod(cls1, "__concat") ==  foo )
true
```

27.3.10 value = class.call(class_table, method, ...)

Calls a method in a given class_table using the given vararg arguments. It throws an error if the 1st parameter is not a class table or if the given method doesn't exist.

```
> cls1,cls1_methods = class("cls1")
> cls1_methods.foo = function(self) print(self.n) end
> class.call(cls1, "foo", { n=5 })
5
```

27.3.11 boolean = class.is_class(class_table)

Returns true/false if the given Lua value is a class table.

```
> cls1,cls1_methods = class("cls1")
> = class.is_class(cls1)
true
```

27.3.12 boolean = class.is_derived(object)

Returns true/false if the given instance object is an instance of a derived class.

```
> cls1,cls1_methods = class("cls1")
> cls2,cls2_methods = class("cls2", cls1)
> o1 = cls1()
> o2 = cls2()
> = class.is_derived(o1)
false
> = class.is_derived(o2)
true
```

27.3.13 class_table,methods_table = class.find(class_name)

Returns the class table associated with the given class_name.

```
> cls1,cls1_methods = class("cls1")
> aux_cls1,aux_cls1_methods = class.find("cls1")
> = ( cls1 == aux_cls1 and cls1_methods == aux_cls1_methods )
```

27.3.14 class.forget(class_name)

Removes the given class_name from the auxiliary table of classes, allowing to redifine this class. **Notice** that the class can't be removed at all because your scripts can have taken the class tables as upvalue, and the instantiated objects will continue working as expected.

```
> cls1 = class("cls1")
> cls1 = class("cls1")
./oop-iter/class.lua:40: cls1 class name exists
stack traceback:
    [C]: in function 'assert'
    ./oop-iter/class.lua:40: in function 'register_class_table'
    ./oop-iter/class.lua:289: in function 'class'
    stdin:1: in main chunk
    [C]: in ?
> class.forget("cls1")
> second_cls1 = class("cls1")
> = ( cls1 == second_cls1 )
false
```

Chapter 28

clustering package

28.1 Introduction

Package `clustering.kmeans.matrix`, available in Lua with `require("aprilann.clustering.kmeans.matrix")`.

The package `clustering` is developed to contain different clustering implementations. Currently, only one is available.

28.2 Package `clustering.kmeans.matrix`

This package contains an implementation of k-means clustering designed to be very efficient with large databases (as large as it could be contained in your main memory), but with a small number of clusters.

28.2.1 `distortion,centroids = clustering.kmeans.matrix{ ... }`

This is the main function of the clustering algorithm. It receives a table with different input arguments, and depending on them, the algorithm could be specialized. The function returns a number with the distortion of the clustering result, and a `matrix` with the centroids (ordered by rows).

28.2.1.1 Starting from a set of known centroids

In this case, the function receives a `matrix` with the initial set of centroids. The given table argument must contain the following fields:

- `data`: it is a `matrix` where the data points are ordered by rows.
- `centroids`: it is a `matrix` with the initial value of the centroids, oredered by rows.
- `max_iter=100`: the maximum number of clustering iterations, by default it is `max_iter=100`.
- `threshold=1e-05`: the threshold for stopping clustering algorithm, by default it is `threshold=1e-05`.
- `verbose=false`: a boolean indicating verbosity.

The algorithm is executed and the given centroids `matrix` will be updated with the newer centroids, resulting from the clustering algorithm.

```
> data = matrix(5,2):linear()
> clusters = matrix(2,2,{0,0,  1,1})
> = clusters
 0           0
 1           1
# Matrix of size [2,2] in row_major [0x1131c30 data= 0x1131d00]
> res, C = clustering.kmeans.matrix{
    data = data,
    centroids = clusters
  }
> = res
3.999998196261
> = C
 1           2
 6           7
# Matrix of size [2,2] in row_major [0x1131c30 data= 0x1131d00]
```

28.2.1.2 Starting from scratch

In this case, the algorithm is implemented in two parts, first the **refine** algorithm published by P.S. Bradley and U.M. Fayyad is used to initialize the centroids `matrix`. After that, the standard clustering algorithm will be used. The given table argument must contain the following fields:

- `data`: it is a `matrix` where the data points are ordered by rows.
- `K`: a number indicating *how many* clusters you want to compute using refine algorithm.
- `random`: a `random` object used by the refine algorithm.
- `subsamples=10`: how many random subsamples of the data will be used in the refine algorithm, by default it is `subsamples=10`.
- `percentage=0.01`: a percentage of the data used by refine algorithm, by default it is `percentage=0.01`, that is, a 1%.
- `max_iter=100`: the maximum number of clustering iterations, by default it is `max_iter=100`.
- `threshold=1e-05`: the threshold for stopping clustering algorithm, by default it is `threshold=1e-05`.
- `verbose=false`: a boolean indicating verbosity.

The algorithm is executed and a centroids `matrix` will be returned.

```
> data = matrix(5,2):linear()
> res,C = clustering.kmeans.matrix{
    data = data,
    K = 2,
    random = random(1234),
  }
> = res
3.999998196261
> = C
 2           3
 7           8
# Matrix of size [2,2] in row_major [0x25d1180 data= 0x25d1250]
```

28.2.2 score,T=clustering.kmeans.matrix.find_clusters(X,C [,T [,verbose]])

This function classifies `X` (a `matrix` with data) in the closest centroid `C` (a `matrix` with the centroids), and returns the `score` of the classification and the tags `T` (a `matrixInt32`). The function receives positional arguments:

1. `X`: a `matrix` with the data ordered by rows (N rows, D columns).
2. `C`: a `matrix` with the centroids ordered by rows (K centroids).
3. `T`: a `matrixInt32` with size Nx1, which contains for every row of `X` the number of its closest centroid. This argument is optional, if not given, a new `martrixInt32` will be allocated.
4. `verbose=false`: a boolean indicating if verbosity is desired. By default it is `false`

```
> = data
 0           1
 2           3
 4           5
 6           7
 8           9
# Matrix of size [5,2] in row_major [0x1717960 data= 0x1714f10]
> score,T = clustering.kmeans.matrix.find_clusters(data,C)
> = score
-51.4
> = T
          1
          1
          1
          2
          2
# MatrixInt32 of size [5,1] in row_major [0x2862a70 data= 0x282e830]
```

Chapter 29

knn package

29.1 Introduction

The package `knn` contains algorithms to deal with K-Nearest-Neighbors algorithm. Currently, an implementation based on k-d tree is available, allowing to work with large databases (as far as it could be loaded into main memory), but with low dimensionality.

29.2 Class `knn.kdtree`

This class is the basic implementation of the k-d tree for KNN classification. It allows to work with matrices of data (as many matrices as you want). After all the data is given, the tree is **built**. No insertion or remove operations are implemented, but it is possible to insert new data and build it again if needed. After the tree is ready, it is possible to query for the nearest-neighbor or the K-nearest-neighbors.

29.2.1 Constructor: `kdt = knn.kdtree(D,random)`

The constructor receives two values, a number `D` with the number of dimensions (columns) of your data, and a `random` object (used at build method).

```
> D = 6
> kdt = knn.kdtree(D,random(9248))
> = kdt
instance 0x214ad70 of knn.kdtree
```

29.2.2 `kdt = kdt:push(matrix)`

This method receives a `matrix` with data ordered by rows, and returns the caller `kdtree` object. The `matrix` pointer is retained by the `kdtree`, but it is not inserted into the structure, the **build** method is who will perform the insertion.

```
> rnd = random(1234)
> m1  = matrix(100,D):uniformf(-1,1,rnd)
> m2  = matrix(200,D):uniformf(-1,1,rnd)
> kdt:push(m1)
> kdt:push(m2)
```

29.2.3 kdt = kdt:build()

This method processes all the given data matrices, and builds the k-d tree structure.

```
> kdt:build() -- after that, it is ready to queries
```

29.2.4 id,distance = kdt:searchNN(matrix)

This method allows to query a *built* `kdtree` object for the nearest-neighbor. It receives a bi-dimensional `matrix` with size `1xD`, and returns two values:

- `id` is the position of the nearest-neighbor. If you take all the *pushed* matrices ordered by rows, this number is the `row` corresponding to the sample in the concatenated data.
- `distance` is a number with the square of the euclidean distance.

```
> id,dist = kdt:searchNN(matrix(1,D):uniformf(-1,1,rnd))
> = id
26
> -- the 26 is located at the first matrix
> = dist
0.49883383064235
> id,dist = kdt:searchNN(matrix(1,D):uniformf(-1,1,rnd))
> = id
178
> -- the 178 is located at the second matrix
> = dist
0.3402419188674
```

29.2.5 point,matrix = kdt:get_point_matrix(id)

This method receives a number `id` of a point in the `kdtree` object (as returned by `kdt:searchNN(...)` method), and returns a `point` which is a `matrix` of size `1xD` with the corresponding point data, and the `matrix` object where the point is contained. Be careful, this method returns a reference to the original data, any change in the data will led to unexpected behavior of the `kdtree` object.

```
> NN = kdt:get_point_matrix(id)
> = NN
 0.657501   -0.604099    0.426221   -0.421949   -0.32904     0.75809
# Matrix of size [1,6] in row_major [0xc80410 data= 0xc754b0]
> = m2(id-100,':')
 0.657501   -0.604099    0.426221   -0.421949   -0.32904     0.75809
# Matrix of size [1,6] in row_major [0xc7ceb0 data= 0xc754b0]
```

29.2.6 result = kdt:searchKNN(K,matrix)

This method performs the K-Nearest-Neighbors search, using the given `K` as number of neighbors and the given `matrix` (with size `1xD`) as data point. The method returns a table with pairs `id,distance`.

```
> result = kdt:searchKNN(4,matrix(1,D):uniformf(-1,1,rnd))
> = result
table: 0x197caa0
> for i=1,#result do print(result[i][1], result[i][2]) end
152 0.42534565841526
40  0.54756417013329
101 0.5931407024824
166 0.66157509210318
```

29.2.7 class = knn.kdtree.classifyKNN(result, get_class_function)

This method receives the result of `kdt:searchKNN(...)` method and a function which transforms a pattern `id` in a class. All the result pairs are traversed, computing the corresponding class from each `id`. The majority vote class will be returned. Normally, the `get_class_function` will be a function which looks into a `dataset`, a Lua table, or a `matrix`, looking for the class of the corresponding `id` number. In some tasks, because of the order of the data, it is possible to compute the class with a math operation. It depends on your data and your experiment framework how to implement this function.

```
> best_class = knn.kdtree.classifyKNN(result,
                                      function(id) return id%10 end)
> = best_class
2
```

29.2.8 table,cls,best = knn.kdtree.posteriorKNN(result, get_cls_func)

This method receives the result of `kdt:searchKNN(...)` method and a function `get_cls_func` which transforms a pattern `id` in a class. All the result pairs are traversed, computing the class posterior probability. A table with pairs of `{class,log posterior}` is returned. The posterior is computed considering the negative of euclidean squared distances as log-scores, and normalizing over all the log-scores in the `result` table. In theory, as more `K` neighbors were taken, the better posterior will be obtained. However, in practice, with values of `K` between 10 and 20 could be enough. **Note** that the posteriors table is traversed using `pairs`, not `ipairs`, because two reasons: first, the class identifier could be anything, not only a number, it depends in your `get_class_function`; and second, even with numeric class, identifiers not all the classes has to be present in the `result` table, so the posteriors table is not an array, it could contains gaps.

Additionally to the `posteriors` table, this function returns the maximum posterior `cls` class and its value `best`.

```
> result = kdt:searchKNN(20,matrix(1,D):uniformf(-1,1,rnd))
> posteriors,bestcls,best = knn.kdtree.posteriorKNN(result,
                                                    function(id) return id%10 end)
> for c,p in pairs(posteriors) do print(c,p) end
1   -3.3052420168408
2   -2.7092774252334
4   -2.289865236587
5   -2.4045060888367
6   -1.8979527591223
7   -2.0151971414884
8   -2.0497985519464
0   -2.2390630830692
9   -1.6785405659992
> -- in this example, the best class is 9, has the maximum posterior
> print(bestcls,best)
9   -1.6785405659992
```

29.2.9 predicted,logp = knn.kdtree.regressionKNN(result, get_tgt_func)

This function receives a `result` table and a `get_tgt_func`, and computes a prediction which is a weighted mean of the neighbors in `result`, weighted by the posteriors computed in a similar fashion as `posteriorsKNN` function, but ignoring the marginalization over classes.

The `get_tgt_func` is a Lua function which receives a training pattern `id` and returns the target (predicted) value associated with it. The returned value must be an instance of `matrix`, a Lua `table` or a Lua `number`. In any case, the `predicted` value will be a `matrix`.

Additionally, this functions returns the the `logp` with the log-posterior of every neighbor.

```
> function price(id)
    -- DO_STUFF
    return price_of_id
  end
> predicted = knn.kdtree.regressionKNN(result, price)
```

Chapter 30

Hyperparameter Optimization tool

30.1 Introduction

Currently, the most widely used hyperparameter optimization technique is grid search. Recently, random search is proposed as an easy method which could obtain interesting good results (competitive with grid search, even better in some tasks) 2012 Bergstra and Bengio.

30.2 Random search hyperparameter optimization

In APRIL-ANN it is possible to do random search hyperparameter optimization using the script located at:

`tools/trainable/random-search-hyperparemeter-optimization.lua`

This scripts receives a configuration Lua file like this:

```
return {
  hyperparams = {
    { option="--o1=",  value=10,  tag="o1", sampling="fixed", hidden=true },
    { option="--o2=",  value=20,  tag="o2", sampling="fixed" },
    { option="--r1",   tag="r1", sampling = "log-uniform",
      type="real", prec=3,
      values= { { min=0.001, max=1 }, },
      filter = function(hyperparams) hyperparams["r1"] = "200" return true end },
    { option="--r2=", tag="r2", sampling = "uniform", values= { 1, 2, 3, 4, 5 } },
    { option="--r3=", tag="r3", prec=3,
      sampling = "gaussian", values= { mean=0, variance=0.1 } },
    { option="--r4=", tag="r4", sampling = "random",
      filter = function(hyperparams)
    if hyperparams["r2"] == "1" then hyperparams["r4"] = "0" end return true
      end },
    { option=nil, tag="r5", sampling="random" }
  },
  filter = function(hyperparams) hyperparams['r5'] = '0.4' return true end,
  script = "",
  exec   = "echo",
  working_dir = "/tmp/",
  -- seed = ANY_SEED_VALUE (if not given, take random from bash)
  n = 50 }
```

The configuration file returns a Lua table which contain some prior knowledge about each hyperparameter (a fully random optimization is unreliable). The table has this major fields:

- `hyperparams`: a table which describes the prior knowledge of each random searched hyperparameter (note that some of them could be 'fixed' instead of random). Each random hyperparemter is identified by a `tag` string, a unique `option` and fields which describe different prior distributions of hyperparameters. The `sampling="fixed|uniform|log-uniform|gaussian|random"` field indicates if the sampling distribution will be fixed (always the same value), uniform, log-uniform, gaussian, or totally random (this last one is not contrained). The fixed distribution needs a `value=SOMETHING` field which contains the value of this hyperparameter. The uniform distribution needs a `values` field which contains a table of values (`values={1, 4, 8}`) or an array of tables with min/max/step constrains (`values={ {min=0, max=10, step=2}, {min=20, max=30, step=4} }`). The log-uniform distribution needs a table with min/max constrains (not step). The field `type="real|integer"` is only useful for min/max/step values. The field `prec=3` indicates the number of precission digits needed. All of them could be `hidden=true`, indicating that this hyperparameter won't be at the output filename string, but yes at the arguments list. Besides, the 'option' field could be `option=nil`, indicating that this hyperparameter is a metaparameter which won't be passed as argument to the script, but yes to the filter functions of each hyperparameter and the global filter function. The `filter` field is a function which returns true or false indicating if this set of hyperparameters is valid, and receives a table indexed by TAGs which contains all top hyperparameter values (it is possible to modify any value at this table).
- `filter`: is a function which received a dictionary table which associates each `tag` with its value (a string in all cases, even for integer or real numbers). This function is called just before run an experiment. It checks the validity of hyperparameters returning true, otherwise, the experiment won't be executed, and it modifies any hyperparameter value. NOTE that is recommended to write filter functions which use 'string' type for their modified hyperparameters.
- `exec`: the executable file, normally an april-ann binary file, but others are possible.
- `script`: it will be an script given as first argument of the executable.
- `working_dir`: where to store stdout of each experiment. Each experiment is stored at a filename "WORKING_DIR/output-TAG1:VALUE1_TAG2:VALUE2_TAG3:VALUE3_..._TAGM:VALUEM.log". Hyperparamteres marked as `hidden=true` won't be used to form this filename.
- `seed`: an optional random number generator seed.
- `n`: the number of experiments which will be executed.

The random search executes a script which receives non positional command line options. The `option` field indicates the string which will be concatenated as prefix of the value. So, if the script needs an option like this: `--blah=NUMBER`, the field may be: `option="--blah="`.

An `option` field could be `nil`, indicating that this hyperparameter is not used at the script, but it is needed in `filter` functions.

WARNING!!! variable params of each filter funtion always has string type, in order to ensure that the required number of precission digits is correct. So, you need to use `tonumber(hyperparam[TAG])` in order to compare two numbers, and also is recommended to modify hyperaparams using string type values.

30.2.1 Command line execution

The execution of the procedure follows this syntax:

```
april-ann tools/trainable/random-search-hyperparemeter-optimization.lua configuration-script.lua
[ARGS]
```

where `ARGS` follows this syntax:

```
ARGS : ARGS [ "all_hyperparams.TAG.value='blah'" | "global_vars.working_dir='blah'" |
"global_vars.n=blah" ... ]
```

where `all_hyperparams` is a Lua table (associates tag names with hyperparmeter fields) which contains the fixed and randomized parameters of `configuration-script.lua`, so it is possible to modify any parameter field (option, value/s, sampling, prec, tag, values.min, values.max, ...) from the command line, and `global_vars` is a Lua table which contains the rest parameters of `configuration-script.lua` (seed, n, exec, script, working_dir, filter). All this command line arguments must be valid Lua code.

Chapter 31

FAQ

1. Is it possible to use a larger bunch_size at validation step?
2. Why is SDAE training stopping after the first layer showing an error output of incorrect matrix dimensions?

31.0.1.0.1 Is it possible to use a larger `bunch_size` at validation step? Yes, it is. A field `bunch_size` could be defined at the table received by `train_dataset` and `validate_dataset` methods of `trainable.supervised_trainer` objects:

```
trainer:train_dataset{
  input_dataset  = in_ds,
  output_dataset = out_ds,
  shuffle        = random_object,
  bunch_size     = 32, -- TRAINING BUNCH SIZE
}
trainer:validate_dataset{
  input_dataset  = in_ds,
  output_dataset = out_ds,
  bunch_size     = 1024, -- VALIDATION BUNCH SIZE
}
```

31.0.1.0.2 Why is SDAE training stopping after the first layer showing an error output of incorrect matrix dimensions? It is a common mistake, probably you forget to use the parameter which is received by `noise_pipeline` functions. See this example:

```
INPUT_DATASET = whatever...
 ...
    noise_pipeline = { function(GIVEN_DS)
                         return dataset.salt_noise{
                                  ds=INPUT_DATASET, ....
                                }
                       end }
 ...
```

This example will produce the error, because the INPUT_DATASET is used inside the function defined for `noise_pipeline` table, and this variable is taken as closure of the function. However, the SDAE procedure

exepcts that you use the GIVEN ARGUMENT `ds`, which has been prepared to contain the data after training the first Auto-Encoder. So, the code must be like this:

```
...
   noise_pipeline = { function(GIVEN_DS)
                        return dataset.salt_noise{
                                 ds=GIVEN_DS, ....
                               }
                      end }
...
```

Chapter 32

LICENSE

- APRIL-ANN, Copyright (c) 2012-2016, ESET, Universidad CEU-Cardenal Herrera, (F. Zamora)
- APRIL-ANN, Copyright (c) 2012-2016, DSIC, Universitat Politècncia de València (S. España, J. Pastor, A. Palacios)
- April, Copyright (c) 2006-2012, DSIC, Universitat Politècnica de València (S. España, J. Gorbe, F. Zamora)

32.1 GNU GENERAL PUBLIC LICENSE

```
GNU GENERAL PUBLIC LICENSE
   Version 3, 29 June 2007
```

Copyright (C) 2007 Free Software Foundation, Inc. http://fsf.org/ Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

```
Preamble
```

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

0. Definitions.

"This License" refers to version 3 of the GNU General Public License.

"Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based on the Program.

To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

A “Standard Interface” means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The “System Libraries” of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A “Major Component”, in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The “Corresponding Source” for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work’s System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users’ Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work’s users, your or third parties’ legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

```
a) The work must carry prominent notices stating that you modified
it, and giving a relevant date.

b) The work must carry prominent notices stating that it is
released under this License and any conditions added under section
7.  This requirement modifies the requirement in section 4 to
"keep intact all notices".

c) You must license the entire work, as a whole, under this
License to anyone who comes into possession of a copy.  This
License will therefore apply, along with any applicable section 7
additional terms, to the whole of the work, and all its parts,
regardless of how they are packaged.  This License gives no
permission to license the work in any other way, but it does not
invalidate such permission if you have separately received it.

d) If the work has interactive user interfaces, each must display
Appropriate Legal Notices; however, if the Program has interactive
interfaces that do not display Appropriate Legal Notices, your
work need not make them do so.
```

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

```
a) Convey the object code in, or embodied in, a physical product
(including a physical distribution medium), accompanied by the
Corresponding Source fixed on a durable physical medium
customarily used for software interchange.

b) Convey the object code in, or embodied in, a physical product
```

```
(including a physical distribution medium), accompanied by a
written offer, valid for at least three years and valid for as
long as you offer spare parts or customer support for that product
model, to give anyone who possesses the object code either (1) a
copy of the Corresponding Source for all the software in the
product that is covered by this License, on a durable physical
medium customarily used for software interchange, for a price no
more than your reasonable cost of physically performing this
conveying of source, or (2) access to copy the
Corresponding Source from a network server at no charge.

c) Convey individual copies of the object code with a copy of the
written offer to provide the Corresponding Source.  This
alternative is allowed only occasionally and noncommercially, and
only if you received the object code with such an offer, in accord
with subsection 6b.

d) Convey the object code by offering access from a designated
place (gratis or for a charge), and offer equivalent access to the
Corresponding Source in the same way through the same place at no
further charge.  You need not require recipients to copy the
Corresponding Source along with the object code.  If the place to
copy the object code is a network server, the Corresponding Source
may be on a different server (operated by you or a third party)
that supports equivalent copying facilities, provided you maintain
clear directions next to the object code saying where to find the
Corresponding Source.  Regardless of what server hosts the
Corresponding Source, you remain obligated to ensure that it is
available for as long as needed to satisfy these requirements.

e) Convey the object code using peer-to-peer transmission, provided
you inform other peers where the object code and Corresponding
Source of the work are being offered to the general public at no
charge under subsection 6d.
```

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A “User Product” is either (1) a “consumer product”, which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, “normally used” refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

“Installation Information” for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User

Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

```
a) Disclaiming warranty or limiting liability differently from the
terms of sections 15 and 16 of this License; or

b) Requiring preservation of specified reasonable legal notices or
author attributions in that material or in the Appropriate Legal
Notices displayed by works containing it; or

c) Prohibiting misrepresentation of the origin of that material, or
requiring that modified versions of such material be marked in
reasonable ways as different from the original version; or

d) Limiting the use for publicity purposes of names of licensors or
authors of the material; or

e) Declining to grant rights under trademark law for use of some
trade names, trademarks, or service marks; or

f) Requiring indemnification of licensors and authors of that
material by anyone who conveys the material (or modified versions of
it) with contractual assumptions of liability to the recipient, for
any liability that these contractual assumptions directly impose on
those licensors and authors.
```

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this

License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An “entity transaction” is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party’s predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

```
                    END OF TERMS AND CONDITIONS

           How to Apply These Terms to Your New Programs
```

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the program's name and a brief idea of what it does.>
Copyright (C) <year>  <name of author>

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see <http://www.gnu.org/licenses/>.
```

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
<program>  Copyright (C) <year>  <name of author>
This program comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type `show c' for details.
```

The hypothetical commands `show w' and`show c' should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see http://www.gnu.org/licenses/.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read http://www.gnu.org/philosophy/why-not-lgpl.html.

32.2 Lua license

Lua originaly is under the terms of MIT license. However the version used here has minimal modifications and is sublicensed as GPL v3.

32.2.1 Lua original License

Lua is licensed under the terms of the MIT license reproduced below. This means that Lua is free software and can be used for both academic and commercial purposes at absolutely no cost.

For details and rationale, see http://www.lua.org/license.html .