October 2, 2002

Torch Tutorial

Ronan Collobert
IDIAP

Contents

CHAPTER 1 Introduction

Torch is a machine learning library, written in C++, which is under a BSD license. When I say "C++", it's in fact C with just the concept of classes found in C++. Therefore, to use ***Torch***, you just need to know C and *the bases* of C++. With the power of C++ classes ***Torch*** has a modular design, and with the power of C[1] the core of ***Torch*** is efficient.

The ultimate objective is to contain all of the state-of-the-art machine learning algorithms, for both static and dynamic problems. Currently it contains all sorts of artificial neural networks (including convolutional network and time-delay neural networks), support vector machines for regression and classification, Gaussian mixture models, hidden Markov models, Kmeans, K nearest neighbors and Parzen windows. It can also be used to train a connected word speech recognizer. And last but not least, bagging and adaboost are ready to use.

In this tutorial, I won't explain you how to install ***Torch*** or how to compile your program: for that, please have a look to the web site[2]. Here I'll try to introduce the main concepts of ***Torch***, and I hope I'll demonstrate that it's easy to use. Note that this document is *not* a reference manual: it's just a tutorial to *start* and to have "the ***Torch*** feeling".

[1] and those of the coders

[2] `http://www.torch.ch`

CHAPTER 2 Coding Guidelines

2.1 The core

The library is divided into several subdirectories. The "foundation" part of ***Torch*** is in `core`. You should never change anything in this directory. If you find a bug, just send me an email. Each main class of algorithm (or concept) is represented by a directory (also called "package"). If you want to do your own algorithm, just create a new package (That is a new directory).

2.2 You said C++ ?

I hate C++. Too much complicated. At the beginning I was thinking about writing the library in Objective C... unfortunately, only few people are using this language. Here are the C++ keywords that you're allowed to use in ***Torch***:

- `namespace` (because ***Torch*** is embedded in the `Torch` namespace).
- `class`
- `virtual`
- `public`
- all C keywords
- `const`, but only if compilers give a warning when you don't use it.
- `new` for `Object` classes. If it's for an array of standard types like `int`, `float`... use `Allocator::alloc()`.

In Torch, every class member are `public`. Sometimes, when you have very obscure and ugly variables in your code, put them in `private`, but that is the only case which is allowed.

I don't want to see:

- `string`... no, no and no, please use `char*`.
- `cin cout`, and C++ streams: no, use `message()`, `warning()`, `error()`, `print()` and `XFile` classes.

- Templates.
- `STL` library.
- Multiple inheritance.
- Operator overload.
- References: use pointers.

As you can see, in ***Torch*** we use C++ only for the object concept, and the heavy `STL` library is banned. The rest of your code should be almost like C.

2.3 Naming conventions

Easy:

- `MyClassName`
- `myMethodName()`
- `my_variable_name`
- `myGlobalFunction()`
- `MY_CONSTANT` (for `#define` constants or other)

Note that each class should be divided into two files: one include file (`.h`, containing the class layout) and one source file (`.cc`, containing the implementation). In the include file, you should provide some documentation, with the `DOC++`[1] format for comments. Moreover, the include file for the class `StupidClass` should look like:

```
#ifndef STUPID_CLASS_INC
#define STUPID_CLASS_INC

/* Your includes which are necessary for this include file here.
   Other includes should be in the implementation file.
*/

namespace Torch {

/* Your definitions here. */

}
#endif
```

[1] See `http://docpp.sourceforge.net/`

CHAPTER 3 Basic Types And Functions

Torch has several types and classes that you should use in your code. You will find these types and classes everywhere in ***Torch*** so you should never forget them.

3.1 The numeric real type

Instead of using `double` or `float`, you should use the `real` type. `real` is defined by the user at compile time, and could be either `float` or `double`. Remember this fact, because all your code *have to work with both type*. For example, some problem could arise if you have to use the `fscanf` function: you have to use the `%lf` string if you're using `double` and `%f` string for `float` mode. To solve this kind of exceptionnal trouble, you can use the `USE_DOUBLE` variable which is *defined* if and only if `real` corresponds to `double`.

3.2 Matrices and vectors

Torch is provided with a **matrix** package. This package defines the `Mat` type for `real` matrices and the `Vec` type for `real` vectors. However you should use this types *only in few cases*, that is, only if you need complex operations on matrices, such as computing the eigenvalues or the inverse of a matrix using the functions of the **matrix** package[1]. Therefore, in ***Torch*** a vector should be a pointer (such as `real *vector`) and a matrix should be a double pointer (such as `real **matrix`). You have been warned[2]...

3.3 Writing message to the user

As explained in the *coding guidelines*, you shouldn't use the C++ streams to write messages to the user. In fact, you shouldn't use the standard function `printf` neither. Please, use the functions:

- `message`: to print a simple message, followed by a carriage return.
- `warning`: to print a warning, followed by a carriage return.

[1] A lot of functions are available in the **matrix** package, but I'll not talk about them here.

[2] Remember that an objective of ***Torch*** is to be *efficient*. So I prefer to see C code which looks like assembly, instead of complex classes coming from a fantasm of a coder and which are only "nice" to see.

- `error`: to print a *fatal* error. The program will end with a call to this function.
- `print`: exactly like `printf`.

All these functions take the same arguments as `printf`. Note that these functions have been created with the perspective that one day, we[3] could want to change the interface of ***Torch***. (For example, to provide a graphical interface).

3.4 *XFile*: the *Torch* streams

When you need to use streams, you should use the `XFile` class. An `XFile` has similar methods than system functions associated to the `FILE` type:

- `read()` reads raw data.
- `write()` writes raw data.
- `eof()` checks if we are at the end of the stream.
- `flushes()` flush buffers of the stream.
- `seek()` seeks to a specified position in the stream.
- `tell()` tells where we are in the stream.
- `rewind()` seeks to the beginning of the file.
- `printf()` prints some text. It has a undetermined number of arguments.
- `scanf()` reads some text. Note that it can read only one variable at each call.
- `gets()` gets a line.

Two additional methods which check what we read or write are provided:

- `taggedWrite(void *, int, int , const char *tag)`: as `write()` but adds in the stream the given tag, and the size of what we are writing.
- `taggedRead(void *, int, int, const char *tag)`: as `read()`, but checks if the given tag is in the stream, and compares the given size to the one stored in the stream with `taggedWrite`.

Currently three kinds of `XFile` are provided: `NullXFile`, which does nothing, `DiskXFile` which corresponds to system `FILE`, and which can handle a file on disk. And `MemoryXFile`, which is a read-write file in memory; this one automatically grows in memory when writing (with a buffered write, to avoid reallocating for one byte).

`DiskXFile` can handle little endian and big endian encoding. For example, you could force all `DiskXFile` to save and load in little endian mode. To know how your processor encodes data, two `static` methods are available: `isLittleEndianProcessor()` and `isBigEndianProcessor()`. To force all `DiskXFile` to use a specific encoding, use

[3] That is, the ***Torch*** team.

`setLittleEndianMode()` and `setBigEndianMode()`. To load and save using the native mode of the processor, (which is the default), use `setNativeMode()`. And to test if you are in the native mode, use `isNativeMode()`. All methods are `static`, which means that all `DiskXFile` will use the same mode. Simple example of use:

```
int main()
{
  DiskXFile::setLittleEndianMode();

  /* Do what you want here. All data will be loaded and saved using
     the little endian mode. If your machine uses big-endian, all
     data will be converted when loading and saving using a DiskXFile.
  */

  return 0;
}
```

`MemoryXFile` uses a list to store its data. And it has a supplementary method: `concat()`. When you call that, you have the garantee that the memory will be concatened (it can be expensive!) in one node of the list. Check the include file for details.

Example:

```
// Create a file on disk named "almost_empty" in write mode.
DiskXFile f_on_disk("almost_empty", "w");

// Create a file in memory (in read write mode)
MemoryXFile f_in_memory;

// Write in binary an array of int
f_in_memory.write(array, sizeof(int), size_of_the_array);

// Write text
f_in_memory.printf("array[0] = %d, array[1] = %d\n", array[0], array[1]);

// Be sure that the memory in only in one node
f_in_memory.concat()

// Ugly write of contents from f_in_memory into f_on_disk
f_on_disk.write(f_in_memory->memory->mem, 1, f_in_memory->size);
```

3.5 Basic classes: *Object* and *Allocator*

All classes in ***Torch*** *must* be children (directly or indirectly) of the class `Object`. This class provides three interesting things: input-output methods for the object, easy option management, and (most importantly) easy memory management.

3.5.1 Input-output methods

Four methods are defined in `Object`:

- `loadXFile(XFile *)` which loads the object from an `XFile` stream.
- `saveXFile(XFile *)` which saves the object into an `XFile` stream.
- `load(char *)` which loads the object from a file on disk with the given name.
- `save(char *)` which saves the object into a file on disk with the given name.

All these methods do nothing by default. Note that these methods don't save or modify the *structure* of the object. It's just for loading or saving the *contents* of the object: the structure must be handled by the user with the constructor of each object.

3.5.2 Option management

An option from an `Object` point-of-view is a variable which can be set by the user at any time after the construction of the object. This means that an option *should not be related to the structure of the object.* There are four types of options which are often used: option for an int, for a real, for a bool, and for a pointer to an Object. The idea is quite simple: in the constructor of the object, the coder uses one of the following methods:

```
addIOption(const char *name, int *ptr,  int init_value,
                                                const char *help="");
addROption(const char *name, real *ptr, real init_value,
                                                const char *help="");
addBOption(const char *name, bool *ptr, bool init_value,
                                                const char *help="");
addOOption(const char *name, Object **ptr, Object *init_value,
                                                const char *help="");
```

('I' for Int, 'R' for Real, 'B' for Boolean and 'O' for Object), and then the user (in a main program) could possibly call one of the corresponding methods:

```
setIOption(const char *name, int option);
setROption(const char *name, real option);
setBOption(const char *name, bool option);
setOOption(const char *name, Object *option);
```

Example: in `Linear.cc` there is a line:

```
addROption("weight decay", &weight_decay, 0, "weight decay");
```

Therefore, to create a Linear layer with a weight-decay equal to 0.001, I just have to write:

```
Linear linear;
linear.setROption("weight decay", 0.001);
```

and that's all. Moreover, I can call `setROption()` at any time to change the weight-decay.

3.5.3 Memory management

In ***Torch*** all memory allocated by a given object should be freed by the *same* object. In fact, the `Object` class contains one special field: `Allocator *allocator`. This object is created in the constructor of `Object` and destroyed in its destructor. An `Allocator` has several methods to allocate memory, and this memory will be automatically freed at the destruction. This means that you have to use `allocator` to allocate memory in an object, and you don't care of the destruction of the memory.

The most interesting methods in `Allocator` are `alloc()`, `realloc()`, which correspond to the system functions `malloc()` and `realloc()`. To create a class which derivates from `Object`, you have to use the overloaded operator new: `new(allocator) YourObject` creates a new `YourObject` and tells that `YourObject` will be destroyed when `allocator` will be destroyed. Note that this new operator couldn't apply to `C++` types like int, real and bool: you really have to use the `alloc()` method for them.

If you want to force the destruction of an allocated memory (or of one object) before the destruction of the allocator, use the method `free()`. If you want that your allocator handles memory that is already allocated and don't belong to it, use `retain()`: the memory will be freed at the destruction. If you want to "steal" the memory from another allocator, use `steal()`. And if you want to stop handling some memory, use `release()`: it means that the allocator won't free the memory at the destruction. It just stops handling it.

Example:

```
int main()
{
  // Creation of allocators
  Allocator *a1 = new Allocator;
  Allocator *a2 = new Allocator;

  // Allocation of memory with a1
  int *array_1 = (int *)a1->alloc(sizeof(int)*10);
  int *array_2 = (int *)a1->alloc(sizeof(int)*10);
  int *array_3 = (int *)a1->alloc(sizeof(int)*10);
  int *array_4 = (int *)a1->alloc(sizeof(int)*10);

  // Create stupid objects with a1
  SupidObject *stupid_1 = new(a1) StupidObject;
  SupidObject *stupid_2 = new(a1) StupidObject;

  // Allocation of an array in a stupid manner
  int *array_5 = (int *)malloc(sizeof(int)*10);

  // a1 takes the control on memory in array_5
  a1->retain(array_5);

  // a2 steals memory in array_2 which belongs to a1
  a2->steal(array_2, a1);

  // a1 doesn't care anymore about memory in array_3
  a1->release(array_3);
```

```
  // Force a1 to free the memory in array_4
  a1->free(array_4);

  // Force a1 to destroy stupid_a2 object
  a1->free(stupid_a2);

  // a1 automatically frees array_1 and array_5 and destroys stupid_1.
  delete a1;

  // a2 automatically frees array_2
  delete a2;

  // Memory leak: array_3 is steal reachable here!
  return 0;
}
```

In fact a main code will usually look like:

```
int main()
{
  /* We usually prefer to allocate an allocator dynamically
     because we prefer to have "new(allocator) StupidObject"
     instead of "new(&allocator) StupidObject".
  */
  Allocator *allocator = new Allocator;

  /*
     Allocate memory here using allocator;
  */

  // Free all memory and destroy all objects
  delete allocator;
  return 0;
}
```

Example in the code of a class:

```
StupidClass::StupidClass()
{
  // Allocate memory using allocator defined in Object
  int *array = (int *)allocator->alloc(sizeof(int)*10);
}

StupidClass::~StupidClass()
{
  // array is automaticaly freed here.
}
```

Note that sometimes, (for hardcore coders) you really need to allocate memory without handling it by an allocator. For that, two `static` methods are available in `Allocator`:

`sysAlloc()` and `sysRealloc()`. These methods correspond to `malloc()` and `realloc()` system functions, except that they print an error if there is no more memory available. Memory allocated with these functions could then be handled by an allocator with the method `retain()`, or freed by the system `free()` function.

3.6 Random functions

In standard C, the `rand` function is available. However, in ***Torch*** you should use only random functions which are described in `Random.h`. `Random` is a class with only static methods providing several kinds of random generators. Before using one of these methods, you can call the `manualSeed()` method which initializes the random generator with a given value. It could be interesting if you want to do several times exactly the same experiment. If you don't do that, a call to the `seed()` method will be automatically done, which initializes the random number generator with the CPU time. In any case, to get the value that *has been used* to initialize the random generator, you can use `getInitialSeed()`.

A lot of random functions are available: check the include file for details!

- `random()` generates a uniform 32 bits integer.
- `uniform()` generates a uniform random number on $[0,1[$.
- `boundedUniform()` generates a uniform random number on $[a,b[$ ($b > a$).
- `normal()` generates a random number from a normal distribution.
- `exponential()` generates a random number from an exponential distribution. The density is $p(x) = \lambda exp(-\lambda x)$, where λ is a positive number.
- `cauchy()` returns a random number from a Cauchy distribution. The Cauchy density is $p(x) = \frac{\sigma}{\pi(\sigma^2+(x-\mu)^2)}$.
- `logNormal()` generates a random number from a log-normal distribution.
- `geometric()` generates a random number from a geometric distribution. It returns an integer i, where $p(i) = (1-p)p^{i-1}$, where p must satisfy $0 < p < 1$.
- `bernouilli()` returns `true` with probability p and `false` with probability $1-p$ ($p > 0$).
- `getShuffledIndices()` returns an array with uniformly shuffled indices.
- `shuffle()` shuffles uniformly a given array.

3.7 Measuring time

When you need to measure CPU time, you can use the `Timer` class. It can be stopped, resumed, reset, and of course you can get the total elapsed time (a `real` in seconds) at any time! Example:

```
int main()
{
  // Create a timer. The timer starts to count now!
  Timer timer;

  /* Do some job1 here */

  // Print the elapsed time for job1
  message("job1: %g CPU seconds", timer.getTime());

  // Stop the timer
  timer.stop();

  /* Do a stupid job here */

  // Resume the timer
  timer.resume();

  /* Do some job2 here */

  // Print the elapsed time for job1+job2
  message("job1+job2: %g CPU seconds", timer.getTime());

  // Reset the timer. As it was not stopped before, the timer
  // counts the time starting from now.
  timer.reset();

  /* Do some job3 here */

  // Print the elapsed time for job3
  message("job3: %g CPU seconds", timer.getTime());

  // Stop the timer.
  timer.stop();

  // Reset the timer. As it was stopped before, the timer
  // doesn't count the time anymore.
  timer.reset();

  /* Do some job4 here */

  // Print "0" we forgot to resume the timer.
  message("job4: %g CPU seconds", timer.getTime());

  return 0;
}
```

3.8 Taking arguments from the command-line

Who said that taking arguments from the command-line was boring ? In ***Torch*** a useful easy-to-use class is provided, and you won't be able to do without it in the near future. Its name is `CmdLine`. This class does the distinction between:

- *arguments* which *must* be present in the command-line.
- *options* which *could* be present in the command-line.

You should never forget that:

- Arguments must be present in the order specified by the pearson who designed the program.
- Options can be completely mixed, but *before* arguments.
- Options are introduced by a keyword specified by the coder. For example, you could have an option `-age` which asks for an integer[4]. To set this this option in the command line, the user will write for example `-age 42`.
- Arguments are not introduced by a keyword. That's why they *must* be in the right order!

Several functions are available to construct the command-line. Some of them only display text when the user needs help. Others add an option or an argument. The text that will be displayed for help depends on the call order of these functions. The methods are:

- `addText(const char *text)` : adds a text line in the help message.
- `info(const char *text)` : adds a text at the beginning of the help. This method should be called only *once*.
- Methods that respectively add an `int`, `bool`, `real` and `char*` option:
  - `addICmdOption(const char *name, int *ptr, int initvalue, const char *help="")`
  - `addBCmdOption(const char *name, bool *ptr, bool initvalue, const char *help="")`
  - `addRCmdOption(const char *name, real *ptr, real initvalue, const char *help="")`
  - `addSCmdOption(const char *name, char **ptr, const char *initvalue, const char *help="")`

  After a call to one of these methods, the option `name` will be added. It means that when you read the command-line, if the option `name` is present, the associated value will be put in `ptr`. Otherwise `initvalue` will be put in `ptr`. Moreover, if the help is called, the `help` message will be displayed. Note that the `-h`, `-help` and `--help` options are reserved. If the user puts one of these options in the command-line, the help will be displayed.

[4]The captain's age

- Methods that respectively add an `int`, `bool`, `real` and `char*` argument:
  - `addICmdArg(const char *name, int *ptr, const char *help="")`
  - `addBCmdArg(const char *name, bool *ptr, const char *help="")`
  - `addRCmdArg(const char *name, real *ptr, const char *help="")`
  - `addSCmdArg(const char *name, char **ptr, const char *help="")`

  The effect is similar to the one obtained with option adding methods. Of course, as the arguments must be present, there is no default value.
- `read(int argc, char **argv)` : read the command-line. Call this method **after** adding options and arguments that you need, with the help of the previous methods.

Here is an example... if I write in my code (of a program named `toy`):

```
const char *help = "It's a stupid toy.";
char *file_in, *file_out;
int n_inputs, max_load;
CmdLine cmd;

cmd.info(help);

cmd.addText("\nArguments:");
cmd.addSCmdArg("file in", &file_in, "the file in");
cmd.addSCmdArg("file out", &file_out, "the file out");
cmd.addICmdArg("n_inputs", &n_inputs, "input dimension of the data");

cmd.addText("\nOptions:");
cmd.addICmdOption("-load", &max_load, -1, "maximum number of examples
                                                          to load");

cmd.read(argc, argv);
```

...the following text will be displayed when the program is lauched without argument, or if I put (for example) a `-h` in the command-line...

```
It's a stupid toy.

 usage: ./toy [options] <file in> <file out> <n_inputs>

Arguments:
  <file in>   -> the file in (<string>)
  <file out>  -> the file out (<string>)
  <n_inputs>  -> input dimension of the data (<int>)

Options:
  -load <int> -> maximum number of examples to load [-1]
```

...and a valid command-line could be:

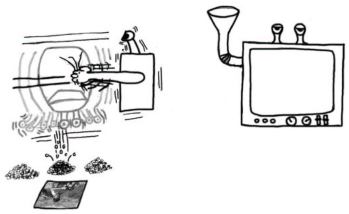
```
./toy -load 666 file_in file_out 42
```

CHAPTER 4 Main Concepts Of *Torch*

There are only four important concepts in ***Torch***. Each of them are implemented in a generic class. And almost all classes of ***Torch*** are subclasses of one of them. Here they are[1]:

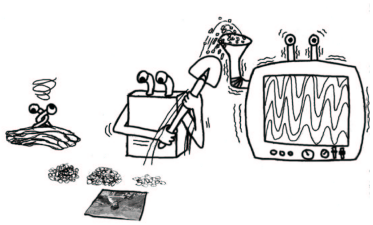
- `DataSet`: this class handles the data. Subclasses could be for static or dynamic data, for data that can fit in memory or on disk, *etc...*
- `Machine`: a black-box that, given an (optional) input and some (optional) parameters, returns an output. It could be for instance a neural network, or a mixture of gaussians.
- `Trainer`: this class is able to train and test a given machine over a given dataset.
- `Measurer`: when given to a trainer, it prints in different files the measures of interest. It could be for example the classification error, or the mean-squared error.

The *general idea* of ***Torch*** is very simple: first, the `DataSet` produces one "training example"...



This training example is given to a machine which computes an output...

[1]Strange pictures are copyrighted ©Nicolas Gilardi 2001. All rights reserved.



...and with that a trainer tries to tune the machine.

As you surely begin to understand[2], for a given machine, you need a special trainer. Usually when you create a new class of *machine learning* machine, you have to write the corresponding trainer. Examples:

- there are many "gradient machines" (`GradientMachine`) (including a *multi-layer perceptron*) which can be trained using a "gradient machine trainer" (`StochasticTrainer`).
- a machine such as a *support vector machine* (`SVMClassification` or `SVMRegression`) can be trained with a trainer which is able to solve a constrained quadratic problem: `QCTrainer`.
- distribution machines (`Distribution`) are usually trained using an *Expectation Maximization* algorithm which is implemented in ***Torch*** with the `EMTrainer`.

We will now have a look in more details on the four ***Torch*** concepts.

[2] I hope.

CHAPTER 5 Data Management

5.1 Sequence

Sequences are used everywhere in ***Torch***. Almost all classes handle sequences. A sequence is a set of real vectors which have the same size. The real vectors are in fact called *frames*. Sequences have a temporal meaning: each frame measures something at a given time. A sequence is defined like this:

```
class Sequence
{
  real **frames;
  int n_frames;
  int frame_size;

  /* methods on sequences */

};
```

Thus, there is `n_frames` in a sequence, each are available in `frames[i]`, which is a real vector of size `frame_size`.

There is many ways for creating a sequence. The common way is to use the constructor

```
Sequence(int n_frames_, int frame_size_);
```

which creates a sequence of `n_frames_` of size `frame_size_`. Values in the frames are in an uninitialized state after this call. A zero number of frames is allowed, but the frame size must be positive.

The second way is to use

```
Sequence(real **frames_, int n_frames_, int frame_size_);
```

Here you specify the array of frames. No copy will be done.

Hardcore coders can use the empty constructor `Sequence()`. But be careful, read the include file in details before, and even the implementation file, in order not to forget any field to fill.

You can do a lot of things on sequences.

- The most important is resizing:

  ```
  resize(int n_frames_, bool allocate_new_frames=true);
  ```

  Usually, don't care about the second argument and forget it. The sequence will be resized to `n_frames_` in an efficient manner: it never forgets the memory allocated for the maximum number of frames you have specified. Hardcore coders may use `allocate_new_frames` (carefully!): if `true`, the frames won't be allocated if the new number of frames if larger than the previous one; you then have to fill `frames` by yourself.

- You can add frames using `addFrame(real *frame, bool do_copy=false)`, or even a whole sequence using `add(Sequence *sequence, bool do_copy=false)`.

- You can do copy from another sequence using `copy(Sequence *from)`, or copy from (or to) a real vector using `copyFrom(real *vec)` and `copyTo(real *vec)`.

- It's also possible to "clone" a sequence: a full copy of the sequence will be returned. This is used in very rare cases, when a class needs to keep in memory sequences which could derivate from `Sequence`, and thus which could have a different structure. Don't care, or if you really need it, check the include file.

5.2 DataSet

5.2.1 Definition

A `DataSet` is an interface to provide a set of *examples*[1]. An example from a ***Torch*** point-of-view is the association of an *input* sequence to a *target* sequence. The key method in a `DataSet` is `setExample(int t)` which sets the `inputs` and `targets` field to the inputs and targets of the example indexed by t. There is `n_examples` examples in a `DataSet`. Thus, the novice programmer can see a `DataSet` as the following:

```
class DataSet
{
  // Number of examples
  int n_examples;

  // Inputs and targets of the setted example
  Sequence *inputs, *targets;

  // Set the example t (0 <= t < n_examples)
  void setExample(int t);
};
```

5.2.2 Dealing with subsets

`DataSet` allows you to deal with subsets. For that, two methods are provided:

[1] Useful in machine learning...

```
void pushSubset(int *subset_, int n_examples_);
void popSubset();
```

`pushSubset()` tells to only consider the `n_examples_` with indices given in `subset_`. `subset_[i]` must be `>= 0` and `< n_examples`. `n_examples` will then be updated to `n_examples_`, and all calls to `setExample(t)` will set the t^{th} selected example. You can do several successive calls to `pushSubset()`: it will do a subset of the previous subset of examples.

After a `pushSubset()` call, when you have finished working on your subset, you have to call `popSubset()` which puts the `DataSet` in its previous state.

If you did a `pushSubset()`, a flag `select_examples` will be set to `true`. At any time (even if there is no pushed subset), there is an array of indices `int *selected_examples` which contains the *real* index of each selected examples. (That is, the index that you would have to give to `setExample()` if no subset was pushed). Moreover, the current *real* index of the set example is given by `real_current_example_index`.

5.2.3 Note on the "existence" of an example

After a `setExample()`, you have the assurance that pointers given in `inputs` and `targets` are valid only until the next `setExample()`:

```
data->setExample(0);
inputs_0 = data->inputs;
targets_0 = data->targets;
data->setExample(1);

/* You cannot play with inputs_0 and targets_0 here.
*/
```

Indeed, some classes which derive from `Dataset` are working on disk, and thus don't keep examples in memory if not specified.

To force an example to be valid after the next `setExample()`, you can use the `pushExample()` (and `popExample()` to forget it) methods. If several examples need to be valid in the same time, you can do several `pushExample()`. They will be valid until the `setExample()` (or `popExample()`) which follows their corresponding `popExample()`. Example:

```
data->setExample(0);
inputs_0 = data->inputs;
targets_0 = data->targets;
data->pushExample();

data->setExample(1);
inputs_1 = data->inputs;
targets_1 = data->targets;
data->pushExample();
```

```
data->setExample(2);

/* inputs_1, targets_1, inputs_0 and targets_0 still valid here.
   inputs and targets in data are those of the example 2.
*/

data->popExample();

/* inputs_1, targets_1, inputs_0 and targets_0 still valid here.
   data->inputs contains inputs_1 and data->targets contains targets_1.
*/

data->popExample();

/* inputs_1 and targets_1 are not valid anymore.
   inputs_0, targets_0 still valid here.
   data->inputs contains inputs_0 and data->targets contains targets_0.
*/

data->setExample(3);

/* inputs_1, targets_1, inputs_0 and targets_0 are not valid anymore.
   inputs and targets in data are those of the example 3.
*/
```

5.2.4 Creating a new DataSet

To create a new `DataSet` which derives from this class, you just have to define `setRealExample()` which sets inputs and targets given the *real* index of the example, and to define `pushExample()` and `popExample()`. Several values must be initialized in the `DataSet`: to achieve that, call the `init()` method in your constructor.

5.3 IOSequence

Before continuing presenting `DataSet`, I need to present the `IOSequence` class, on which most `DataSets` are based. `IOSequence` provides an ensemble of sequences. All these sequences have the same frame size, but could have different number of frames. The class is clearly explained in the include file:

```
class IOSequence : public Object
{
  public:
    // Number of sequences in the interface.
    int n_sequences;

    // Frame size of each sequence.
    int frame_size;

    //
```

```
    IOSequence();

    // Returns the number of frames of the sequence indexed by #t#.
    virtual int getNumberOfFrames(int t) = 0;

    /** Write the sequence #t# in #sequence#.
        Sequence must have the size returned by #getNumberOfFrames()#.
    */
    virtual void getSequence(int t, Sequence *sequence) = 0;

    // Returns the total number of frames in the IO.
    virtual int getTotalNumberOfFrames() = 0;

    virtual ~IOSequence();
};
```

Thus, to get the sequence indexed by t you need three steps:

```
// 1) Get the number of frames of the sequence
int n_frames = io_sequence->getNumberOfFrames(t);

// 2) Allocate a sequence
Sequence sequence(n_frames, io_sequence->frame_size);

// 3) Get the sequence
io_sequence->getSequence(t, &sequence);
```

It's pretty easy to define a new `IOSequence` class: you just have to define pure virtual methods, and to set `n_sequences` and `frame_size` in the constructor.

Several `IOSequence` already exist. In the following paragraphs you'll find a short description of them.

5.3.1 IOAscii

It reads Ascii data on the disk. The file data must have the following format:

```
n m
frame 1 of the sequence (m real)
...
frame n of the sequence (m real)
```

where m and n are integers corresponding respectively to the number of frames and the frame size of the sequence. The constructor is:

```
IOAscii(const char *filename_, bool one_file_is_one_sequence_=false,
                                                    int max_load_=-1);
```

If `one_file_is_one_sequence_` is false, it means that a row of the file will be viewed as a sequence with one frame in the `IOAscii`. Thus, it will contain n sequences. Otherwise,

the `IOAscii` will contain only one sequence with n frames. In any cases, `max_load_` specifies how many *sequences* to load. Note that this `IO` must be accessed in a *sequential* manner. Moreover, the first access opens the file, and the access of the last sequence closes the file.

It's possible to save in a specified stream a sequence in *Ascii* format using the additionnal static method:

```
static void saveSequence(XFile *file, Sequence *sequence);
```

5.3.2 IOBin

The format is the same as `IOAscii`, but in binary. Of course there is no space and carriage return! The constructor is:

```
IOBin(const char *filename_, bool one_file_is_one_sequence_=false,
                             int max_load_=-1, bool is_sequential=true);
```

The functionalities are exactly the same if `is_sequential` is true. If this flag is false, the file can be accessed in a random manner. Note however that the file will be opened and closed at each access and that a "seek" will be applied: this could be slow, depending on you system and on the file size.

5.3.3 IOMulti

It takes an array of `IOSequence` in the constructor and works as if there was only one ensemble of sequences: the number of sequences contained in `IOMulti` will be the total number of sequences contained in all `IOSequence` of the array. All given `IOSequence` must have sequences with the same frame size.

5.3.4 IOSub

It takes an `IOSequence` in the constructor and shows only a subset of adjacent columns in each frame of each sequence when calling `getSequence()`.

5.4 MemoryDataSet and DiskDataSet

Currently, there are two major branches of `DataSet` in ***Torch***: `DataSet` which are fully loaded in memory (`MemoryDataSet`) and `DataSet` where only one example is loaded in memory at each `setExample()` and where the rest stays on disk (`DiskDataSet`). Both of these `DataSet` are designed to be used with the `IOSequence` class[2]. The idea is quite simple: there is one `IOSequence` which provides sequences for inputs, and another for targets. `MemoryDataSet` reads all examples in the constructor through the `IOSequence`, whereas `DiskDataSet` reads one example at each `setExample()` by one call to `IOSequence`.

[2]In fact, it's possible to create a *MemoryDataSet* without them, but it's not recommended.

Creating a new `DataSet` from a `MemoryDataSet` or a `DiskDataSet` is effortless: you just have to give two `IOSequence` (one for inputs and one for targets) to the `init()` method provided in both classes. This method should be called in the constructor of your sub-class, and does all the job for you. Note that you can give a NULL pointer if you don't have any input (or any target).

5.5 MatDataSet and DiskMatDataSet

The standard `DataSet` file format is handled by `MatDataSet` for `MemoryDataSet` and `DiskMatDataSet` for `DiskDataSet`. They have both the same functionalities. Thus, I will focus on `MatDataSet` here. The first constructor is:

```
MatDataSet(const char *filename, int n_inputs_, int n_targets_,
      bool one_file_is_one_sequence=false, int max_load=-1,
                              bool binary_mode=false);
```

The data will be read from `filename` in an `IOAscii` format (or `IOBin` if `binary_mode` is true). Each line in the file is divided into two parts: the first `n_inputs_` correspond to an input frame, and the rest of the line is a target frame. If `one_file_is_one_sequence` is false, the file is viewed as n sequences with one frame of size `n_inputs_` for inputs and `n_targets_` for targets. (Where n is the number of lines in the file). Otherwise it will be considered as one sequence with n frames. If `max_load` is positive, it will load only `max_load` *sequences*[3].

A similar constructor is available:

```
MatDataSet(const char **filenames, int n_files_, int n_inputs_,
  int n_targets_, bool one_file_is_one_sequence=false,
      int max_load=-1, bool binary_mode=false);
```

It has the same behavior as the previous constructor, but the data is divided into `n_files_` given in the array `filenames`.

When you have one file per sequence for inputs, and one file per sequence for targets, it's possible to use the constructor

```
MatDataSet(const char **input_filenames,
           const char **target_filenames,
           int n_files_, int max_load=-1, bool binary_mode=false);
```

Once again, `max_load` specifies the number of input and target *sequences* to load. If `binary_mode` is false, the files must be in the `IOAscii` format (one row is one frame), otherwise they must be in the `IOBin` format.

[3]Thus, it has a meaning only when `one_file_is_one_sequence` is false.

5.6 Pre-processing

It is possible to do pre-processing over sequences of a `DataSet`. Note that this pre-processing must respect the *structure* of the sequences: it can just modify the *content*. A `PreProcessing` class contains two methods to be defined if you need a new kind of pre-processing:

```
// Given an input sequence, do the pre-processing.
void preProcessInputs(Sequence *inputs);

// Given a target sequence, do the pre-processing.
void preProcessTargets(Sequence *targets);
```

These methods modify the content of a given input or target sequence as you want. To ask a `DataSet` to perform a given pre-processing, just call the following method:

```
void preProcess(PreProcessing *pre_processing);
```

Note that for a `MemoryDataSet`, the pre-processing will be done immediatly after a call of this method, for all sequences. For a `DiskDataSet`, the pre-processing will be done "on-the-fly" after each call of `setExample()`.

Currently, only one class of pre-processing is provided: `MeanVarNorm`.

```
MeanVarNorm(DataSet *data, bool norm_inputs=true,
                                   bool norm_targets=false);
```

It takes a `DataSet` in the constructor, and normalizes *all the sequences* contained in this `DataSet` in a way that the variance[4] of the ensemble of all *frames* is 1 for all frame columns, and the mean is 0. In fact, the computation of the normalization is done in the constructor, and the the normalization itself is done when calling `preProcessInputs()` or `preProcessTargets()`. By default, it normalizes only inputs. Here is an example:

```
// Create a data set
MatDataSet data(file, n_inputs, n_targets);

// Computes means and standard deviations
MeanVarNorm mv_norm(&data);

// Normalizes the data set
data.preProcess(&mv_norm);
```

5.7 ClassFormatDataSet

One common problem when you are doing classification in machine learning, is that the class encoding format usually depends on the learning algorithm, and you don't want

[4] We divide by the standard deviation.

to have several versions of your data set on the disk. `ClassFormatDatSet` solves this problem: it takes another `DataSet`, and converts the targets according to a "translation" array.

The targets in the given array *must* have the following format: it is sequences with one frame of size one. The values that they contain *must* be 0 for the first class, 1 for the second class, 2 for the third class, and so on.

Two constructors are available to create such a `DataSet`:

- `ClassFormatDataSet(DataSet *data, Sequence *class_labels)` where you provide the encoding format of each class through `class_labels`. The class i will be view as `class_labels->frames[i]`.
- `ClassFormatDataSet(DataSet *data, int n_classes=-1)` where it is assumed that you want the "one-hot format". The number of classes is specified with `n_classes`, if this one is positive. Otherwise, the number of classes is guessed by looking at the maximum target value in the given `DataSet`.

CHAPTER 6 Rage Against The Machine

A machine is a simple class which contains `Sequence *outputs`, and a method `forward(Sequence *inputs)` which updates `outputs` given `inputs`.

It contains also two other methods:

- `setDataSet(DataSet *data)` which is called before training. Thus, if your machine depends on the training set, and you need to do something on its structure knowing this training set, do it in this method.
- `reset()`: if it makes sense that your machine could be set in a "random" mode, do it here. This method will be called for example when doing cross-validation.

I'll present now the two main classes of machines which are implemented in the packages `gradients` and `kernels` of ***Torch***: `GradientMachine` and `SVM`. Of course other machines are available in ***Torch*** packages, but check the documentation of each package if you want to know more.

6.1 Gradient Machines

6.1.1 Introduction

Gradient machines are machines which could be trained with a gradient descent algorithm. To achieve that, they contain several supplementary fields

```
/* Contains parameters of the machine
   Check include files to know the structure of params.
*/
Parameters *params;

// Contains the derivative of the machine with respect to the parameters
Parameters *der_params;

// Contains the derivatives of the machine with respect to the inputs
Sequence *beta;
```

Then, a method `backward(Sequence *inputs, Sequence *alpha)` is available, which updates `der_params` and `beta`, given the `inputs` and `alpha` which is a derivative

that comes from subsequent machines and criterions. Thus `beta` is valid only after a `backward()` and has the same structure as the given `inputs` (same number of frames and same frame size). And the given `alpha` has also the same structure as `outputs`. Note that you have to *accumulate* the derivatives with respect to the parameters, and *not accumulate* the derivatives with respect to the inputs of the machine. This is because several machines could share the same parameters. (It's the trainer which initializes the derivatives with respect to the parameters to zero).

To simplify the life of the coder, several fields have been added:

- `n_inputs` and `n_outputs` which are the size of input frames and output frames. These fields should be fixed in the constructor.
- As many gradient machines don't use the time aspect of a sequence, two methods have been added:

```
frameForward(int t, real *f_inputs, real *f_outputs);
frameBackward(int t, real *f_inputs, real *beta_,
                                        real *f_outputs, real *alpha_);
```

  These methods are called for each frame by the default `forward()` and `backward()` methods. The first one writes in `f_outputs` the outputs given the frame `f_inputs`. If needed, the index of the frame is given with `t`. The second one updates `beta_`, the derivative with respect to the input frame `f_inputs`. It should also update the derivatives of the parameters contained in `der_params`. When creating a new gradient machine, if your machine don't use the time aspect of sequence, you just have to overload these two methods. Otherwise, you should overload `forward()` and `backward()`. Note that `frameForward()` and `frameBackward()` should never be called outside the class, because there is no warranty that the coder used them!

- `loadXFile()` and `saveXFile()` which load and save the *parameters*.

And last, you should take into account the flag `partial_backprop` when writing a method `backward()` or `frameBackward()`. Indeed, if it's false, the machine shouldn't compute the derivatives `beta` with respect to the inputs. This flag if set by the method `setPartialBackprop()`. You shouldn't have to modify this method.

6.1.2 ConnectedMachine

There exist a lot of small `GradientMachine`. Among them you will find `Linear`, `Tanh`, `Sigmoid`, `Exp`... but the most important one is `ConnectedMachine`. It allows you to plug all other machines as you want, to obtain what you want. Here are the important methods it contains:

```
/* Add a Full Connected Layer. The #machine# is fully connected
   to the previous layer. If necessary, a layer is added before
   adding the machine.
*/
void addFCL(GradientMachine *machine);

// Add a #machine# on the current layer
void addMachine(GradientMachine *machine);
```

```
/* Connect the last added machine on #machine#.
   Note that #machine# \emph{must} be in a previous layer.
*/
void connectOn(GradientMachine *machine);

// Add a layer (you don't have to call that for the first layer)
void addLayer();
```

Example: how could I create a multi-layered perceptron with tanh non-linear hidden units and sigmoid outputs ? Here it is:

```
// Creates the layer necessary to the MLP
Linear layer1(n_inputs, n_hidden_units);
Tanh layer2(n_hidden_units);
Linear layer3(n_hidden_units, n_outputs);
Sigmoid layer4(n_outputs);

// Creates the MLP itself
ConnectedMachine mlp;
mlp.addFCL(&layer1);
mlp.addFCL(&layer2);
mlp.addFCL(&layer3);
mlp.addFCL(&layer4);

/* The previous code is equivalent to:

   mlp.addMachine(&layer1);
   mlp.addLayer();
   mlp.addMachine(&layer2);
   mlp.connectOn(&layer1);
   mlp.addLayer();
   ... and so on...

   Another possibility:

   mlp.addFCL(&layer1);
   mlp.addMachine(&layer2);
   mlp.connectOn(&layer1);
   mlp.addLayer();
   ... and so on...
*/

// Never forget that!
mlp.build();
```

As you can see, it's pretty easy. However, don't forget the call to the `build()` method after constructing your machine: it realizes the connections between machines and check if connections are valid. You should call this method in the constructor if you are doing a sub-class of `ConnectedMachine`.

Note also that several calls to `connectOn()` method for the same machine will add all outputs of specified machines in `connectOn()` at the input of the last added machine. Example:

```
Linear layer1_1(n_inputs, n_hidden_units);
Linear layer1_2(n_inputs, n_hidden_units);
Tanh layer2_correct(2*n_hidden_units);
Tanh layer2_wrong(n_hidden_units);

// The following code is correct:
ConnectedMachine mlp;
mlp.addMachine(&layer1_1);
mlp.addMachine(&layer1_2);
mlp.addLayer();
mlp.addMachine(&layer2_correct);
mlp.connectOn(&layer1_1);
mlp.connectOn(&layer1_2);
mlp.build();

/* But this one is false:
  ConnectedMachine mlp;
  mlp.addMachine(&layer1_1);
  mlp.addMachine(&layer1_2);
  mlp.addLayer();

  // layer2_wrong don't have the right number of inputs
  mpl.addMachine(&layer2_wrong);
  mlp.connectOn(&layer1_1);
  mlp.connectOn(&layer1_2);

  // build will do an error here
  mlp.build();
*/
```

6.2 SVM for dummies

In ***Torch*** SVM classes derive from a class called `QCMachine` (for Quadratic Constrained Machine) which is a little bit more general than SVM. Two classes are provided: `SVMClassification` and `SVMRegression`. Both require a `Kernel` in argument. This is a class which has a method `eval()` which evaluates the kernel given two sequences. In fact, kernels provided in the core of ***Torch*** use only the first frame of each sequence, but everything is possible... To create a new kernel class, you just have to define the method `eval()`.

Provided kernels are:

- `DotKernel`, the dot product kernel.
- `PolynomialKernel`, which computes $k(x, y) = (sx.y + r)^d$.
- `GaussianKernel`, $k(x, y) = e^{-\gamma\|x-y\|^2}$.

- `SigmoidKernel`[1], $k(x, y) = tanh(s * x.y + r)$.

I will focus now on SVM for classification, but the code is similar to SVM for regression. Note that SVM needs to minimize the following quadratic function:

$$Q(w, b) = 0.5 * \|w\|^2 + \sum_j C_j |1 - y_j * (w.x_j + b)|_+$$

(This is when using a dot-product. Otherwise we send data in a feature space using a kernel). (x_j, y_j) are the training examples and C_j is the trade-off between the margin and the error for example j. In classification, targets y_j *must be* $+1$ or -1.

Then to create an SVM for classification, for example, it's pretty basic:

```
// Choose a kernel
GaussianKernel kernel(gamma);

// You made it!
SVMClassification svm(&kernel);
```

If you need it, you can specify the weights C in an optional parameter, but the size of this array *must correspond* to the *real* number of examples of the data set you will use to train the SVM. Usually, don't care, you will only set the option `"C"` using `setROption()`: it will set all C_j to the given value. You could also set the real option `"cache size"`: the larger the better (faster). It's the size in mega bytes of the cache used to store the most used parts of the kernel matrix, during training.

[1] That's not really a kernel, because it doesn't satisfy Mercer conditions.

CHAPTER 7 Measurers

7.1 Introduction

`Measurers` are strange entities which can measure anything you have in mind. They are usually automatically called during training and testing phases. A `Measurer` needs two things to live:

- a `DataSet`. Even if it doesn't use it for measuring, the `Trainer` uses it to know when to call the measurer. Remember that!
- a `XFile`. The measurer should output all its measures to this file. If the user doesn't want (for some special reason) to specify one at the construction, he can pass one `NullXFile`.

The measurer is equipped with four methods:

- `measureExample()` which should be called by the `Trainer` after doing a `forward()` for each example (or another similar method, if this one doesn't make sense).
- `measureIteration()` which should be called after having seen all examples.
- `measureEnd()` which should be called at... the end! It could be the end of training or testing phase.
- `reset()` called once by the `Trainer` before the training or testing phase.

The interpretation of `measureIteration()` and `measureEnd()` could differ slightly from one measurer to another.

Note also that measurers have an option called `"binary mode"` which specifies how they should encode their outputs in their `XFile` stream. There are a lot of measurers, so I'll focus on only two examples here, to show how you could use them.

7.2 MSEMeasurer

This measures the mean-square error between the current targets of a `DataSet` and the given input sequence. Example: suppose you want to measure the MSE during training of a machine called `mlp` on the training data set `data`:

```
// We will put results in a file on disk
DiskXFile mse_file("the_mse_error", "w");

// Create it
MSEMeasurer mse_meas(&mlp.outputs, &data, &mse_file);

/* In fact, as you will see in the next chapter, Trainer
   take a list of measurers...
   Add this measurer to a list.
*/
MeasurerList measurers;
measurers.addNode(&mse_meas);

// Now, give this list to the Trainer...
```

Note also that `MSEMeasurer` has several options to normalize by the number of examples, of frames and by the frame size.

7.3 ClassMeasurer

It measures the classification error of the given inputs compared to the current targets of a `DataSet`. However, there are many ways to encode a class: therefore, in ***Torch*** there is a class called `ClassFormat` for specifing this format. Three formats are already proposed:

- `MultiClassFormat`: the classes are encoded with *one real* for each class (usually it's an integer like 0, 1, ...).
- `TwoClassFormat`: like `MultiClassFormat`, but it outputs an error if it detects more than two classes. It has been included, because a lot of algorithms work only with two classes.
- `OneHotClassFormat`: given a vector v encoded in "one-hot" format, $v_i = 1$ if v corresponds to the class i, else $v_i = 0$. Concrete example: if it has three classes, the first one will be encoded with 100, the second with 010, and the last with 001.

A `ClassMeasurer` takes a `ClassFormat` and a `DataSet`. It assumes that the given inputs and the targets of the `DataSet` *are encoded in the same specified format.*

Example: suppose you want to measure the class error during training of a machine called `mlp` on the training data set `data`. The `DataSet` has a target frame size equal to `n_targets`, and they are encoded in "one-hot" format. The `mlp` has `n_targets` outputs.

```
// We will put results in a file on disk
DiskXFile class_file("the_class_error", "w");

// Create the encoding format
OneHotClassFormat class_format(n_targets);

// Create it
```

```
ClassMeasurer class_meas(&mlp.outputs, &data, &class_format, &class_file);

/* In fact, as you will see in the next chapter, Trainer
   take a list of measurers...
   Add this measurer to a list.
*/
MeasurerList measurers;
measurers.addNode(&class_meas);

// Now, give this list to the Trainer...
```

Note also that `ClassMeasurer` has two optional parameters in the constructor to specify if it prints the confusion matrix or not. (At each iteration or when calling `measureEnd()`).

CHAPTER 8 Train The Beast

A `Trainer` takes a machine in its constructor, and is able to train and test it.

```
void train(DataSet *data_, MeasurerList *measurers);
void test(MeasurerList *measurers);
```

During the train phase, you have the possibility of given a list of `Measurer`. The trainer should have to call them during training. To test the machine, use the `test()` method. The trainer knows on which `DataSet` to call the `Measurer`, because each `Measurer` is associated with a `DataSet`.

As the core of ***Torch*** contains two kinds of machine (SVM and gradient machines), there are two kinds of `Trainer` for these machines: `StochasticGradient` for `GradientMachine` and `QCTrainer` for `SVM`. I'll focus on these classes now, and then I'll introduce ensemble models such as `Bagging`, `Adaboost` and `KFold`.

8.1 StochasticGradient

This trainer takes a `GradientMachine` and a `Criterion` at construction. A `Criterion` is a class designed for this `Trainer`: it gives the cost function to be used to train the given machine. A `Criterion` has an output with a frame size of one. The output gives an error, which can be taken into account by the `Trainer` to stop training, if this error tends to be constant. A criterion is also able to backpropagate this error. Finally, a `Criterion` has a `DataSet` field named `data`, set by the trainer before training (on the training set) using the method `setDataSet()`. It can be useful to compute the error!

Several criterion are given as standard:

- `MSECriterion`: computes the MSE between the inputs of the criterion and the `targets` in its associated `DataSet`.
- `ClassNLLCriterion`: a criterion designed to train a gradient machine via a maximization of the *negative log-likelihood*. Input i of the criterion should be the log-probability for class i. This `Criterion` needs a `ClassFormat` at the construction, in order to know the target encoding format of the training `DataSet`.
- `WeightedMSECriterion`: as for `MSECriterion`, but you can give a weight for each example of the training `DataSet`. Note that the size of the weight vector *must* correspond to the number of *real* example in the training `DataSet`.

- `MultiCriterion`: this is a `Criterion` which is a weighted sum of other `Criterion`. It takes an array of `Criterion`. By default, the weights are equal to one.

Given a `Criterion`, `StochasticGradient` will train a `GradientMachine` using a stochastic gradient descent algorithm. It has several options:

- `"end accuracy"`: if the difference between the previous error and the current error given by the `Criterion` is less than the given value, stop training. By default, this is set to 0.0001.
- `"learning rate"`: if λ is the given value, the update of a weight w will be $w^{t+1} = w^t - \lambda * \frac{\delta C}{\delta w^t}$. Set to 0.01 by default.
- `"learning rate decay"`: if μ is the given value, the learning rate λ_i used at *iteration* i (that is after having seen all examples i times) will be $\lambda_i = \frac{\lambda}{1+i\,\mu}$. Default value is 0.
- `"max iter"`: if the number of iterations is equal to the given number, training will end. Default value is 25.
- `"shuffle"`: if `true`, use shuffled indices to access the training examples. `true` is the default value.

Note also that the `Measurer` associated with the training set will compute the error *on the fly*.

Example: load a `DataSet`, train a MLP for 50 iterations using maximization of the negative log-likelihood, and print the train error in a file.

```
// Create a data set from 'file'
MatDataSet data(file, n_inputs, n_targets);

// Normalize it!
data.normalize();

// Create a MLP with LogSoftMax outputs
ConnectedMachine mlp;

Linear layer1(n_inputs, n_hidden_units);
Tanh layer2(n_hidden_units);
Linear layer3(n_hidden_units, n_targets);
LogSoftMax layer4(n_targets);

mlp.addFCL(&layer1);
mlp.addFCL(&layer2);
mlp.addFCL(&layer3);
mlp.addFCL(&layer4);

// Build it!
mlp.build();

// Just to be faster. Not required.
mlp.setPartialBackprop();
```

```
// Use one-hot class format
OneHotClassFormat class_format(&data);

// The classification measurer: output results in 'the_class_error'
DiskXFile out("the_class_error", "w");
MeasurerList measurers;
ClassMeasurer class_meas(mlp.outputs, &data, &class_format, &out);
measurers.addNode(&class_meas);

// The Negative Log-Likelihood criterion
ClassNLLCriterion nllc(&class_format);

// The trainer
StochasticGradient trainer(&mlp, &nllc);
trainer.setIOption("max iter", 50);

// Train it!
trainer.train(&data, &measurers);
```

8.2 QCTrainer

This is designed to train `QCMachine`, and thus it can train an `SVM`. It has several options:

- `"iter shrink"`: the number of iterations before trying to shrink variables which seem to be fixed at bounds. After shrinking, the trainer will wait this number of iterations before trying to shrink again. The default value is 100, but you should tune this option, especially when using noisy data set. To deactivate shrinking, give a large number here.
- `"end accuracy`: the trainer will stop if all non-shrinked variables verify the KKT conditions with the given accuracy. Usually, the default value (0.01) works well.
- `"iter message"`: gives the number of iterations between printing a message of the current number of active variables and the maximum KKT error.
- `"unshrink"`: if false, the trainer will test at the end of training if the shrinked variables satisfy the KKT conditions. If not, it will unshrink variables and continue training. It is not encouraged to set this value to true. Instead, put a large value in `"iter shrink"` in order to deactive shrinking.
- `"max unshrink"`: if unshrink is set, the train will do unshrinking at most the given number of times. After that, shrinking will be deactived.

Note that all `Measurer` given to `QCTrainer` during the training phase will be ignored. You can test the machine only using the `test()` method.

Example:

```
/* Create a data set from 'file'
   As we want to do classification with SVM, the target values
```

```
   must be -1 or 1.
*/
MatDataSet data(file, n_inputs, 1);

// Choose a kernel
GaussianKernel kernel(gamma);

// You made it!
SVMClassification svm(&kernel);

// Trade-off error/margin
svm.setROption("C", 100);

// Cache size: 100Mo
svm.setROption("cache size", 100);

// The trainer...
QCTrainer trainer(&svm);

// Train the svm!
trainer.train(&data, NULL);
```

8.3 Bagging And Boosting

`Bagging` and `Boosting` are `Trainer` implemented in a similar manner. Thus I will focus only on the `Bagging` class here. These classes take in the constructor of a `WeightedSumMachine`. Essentially, a `WeightedSumMachine` takes an array of `Trainer`, and possibly some weights, and is able to calculate (in its `forward()` method) the weighted sum of the ouput of each machine associated with each trainer, given some inputs. The `Boosting` trainer will optimize the weights of this machine.

Doing bagging can be divided in four steps:

- Create `n_bag` trainers (associated to `n_bag` machines), where `n_bag` is the number of bags you want:

```
MLP **mlp = (MLP **)allocator->alloc(sizeof(MLP *)*n_bag);
StochasticGradient **trainer = (StochasticGradient **)
             allocator->alloc(sizeof(StochasticGradient *)*n_bag);

for(int i = 0; i < n_bag; i++)
{
  mlp[i] = new(allocator) MLP(n_inputs,n_hu,n_targets);

  trainer[i] = new(allocator) StochasticGradient;
  trainer[i]->setIOption("max iter", max_iter);
  trainer[i]->setROption("end accuracy", accuracy);
  trainer[i]->setROption("learning rate", learning_rate);
  trainer[i]->setROption("learning rate decay", decay);
}
```

- Create a `WeightedSumMachine`. You have the possiblility of giving an array of measurers which will be called during training of each machine:

```
WeightedSumMachine bag_machine(trainer, n_bag, NULL);
```

- Create the bagging trainer:

```
Bagging bagging(&bag_machine);
```

- Train...

```
// Suppose we want to measure the MSE of the bagging...
DiskXFile out("the_mse_error", "w");
MSEMeasurer mse_meas(bag_machine.outputs, &data, &out);
MeasurerList measurers;
measurers.addNode(&mse_meas);

// Go...
bagging.train(&data, &measurers);
```

8.4 KFold

The `KFold` class is not a `Trainer`, but an `Object` which takes a `Trainer`. As it doesn't correspond to any other category, I will present it here. It provides an interface to sample data, for use by methods such as cross-validation. It's pretty simple to use: just provide a `Trainer` and the number of folds

```
KFold(Trainer* trainer_, int kfold_);
```

Then call the method

```
crossValidate(DataSet *data, MeasurerList *train_measurers=NULL,
                                  MeasurerList *test_measurers=NULL,
                               MeasurerList *cross_valid_measurers=NULL);
```

Where `train_measurers` are called in each "train pass" for each fold, `test_measurers` are called in each "test pass" for each fold, and `cross_valid_measurers` are called during the "cross-validation loop".

If you need to, you can redefine the method `sample()` which by default provides samples for the standard cross-validation.