Torch7: A Matlab-like Environment for Machine Learning

Ronan Collobert[1] **Koray Kavukcuoglu**[2] **Clément Farabet**[3,4]

[1] Idiap Research Institute
Martigny, Switzerland

[2] NEC Laboratories America
Princeton, NJ, USA

[3] Courant Institute of Mathematical Sciences
New York University, New York, NY, USA

[4] Université Paris-Est
Équipe A3SI - ESIEE Paris, France

Abstract

Torch7 is a versatile numeric computing framework and machine learning library that extends Lua. Its goal is to provide a flexible environment to design and train learning machines. Flexibility is obtained via Lua, an extremely lightweight scripting language. High performance is obtained via efficient OpenMP/SSE and CUDA implementations of low-level numeric routines. Torch7 can easily be interfaced to third-party software thanks to Lua's light interface.

1 Torch7 Overview

With Torch7, we aim at providing a framework with three main advantages: (1) it should *ease the development of numerical algorithms*, (2) it should be *easily extended* (including the use of other libraries), and (3) it should be *fast*.

We found that a scripting (interpreted) *language with a good C API* appears as a convenient solution to "satisfy" the constraint (2). First, a high-level language makes the process of developing a program simpler and more understandable than a low-level language. Second, if the programming language is *interpreted*, it becomes also easier to quickly try various ideas in an interactive manner. And finally, assuming a good C API, the scripting language becomes the "glue" between heterogeneous libraries: different structures of the same concept (coming from different libraries) can be hidden behind a unique structure in the scripting language, while keeping all the functionalities coming from all the different libraries.

Among existing scripting languages[1] finding the ones that satisfy condition (3) severely restricted our choice. We chose Lua, the fastest interpreted language (with also the fastest Just In Time (JIT) compiler[2]) we could find. Lua as also the advantage to have been designed to be easily *embedded* in a C application, and provides a *great* C API, based on a virtual stack to pass values to and from C. This unifies the interface to C/C++ and makes library wrapping trivial.

Lua is intended to be used as a powerful, light-weight scripting language for any program that needs one. Lua is implemented as a library, written in clean C (that is, in the common subset of ANSI C and C++). Quoting Lua webpage[3],

[1] For e.g. on http://shootout.alioth.debian.org.
[2] http://luajit.org/
[3] http://www.lua.org.

> Lua combines simple procedural syntax with powerful data description constructs based on associative arrays and extensible semantics. Lua is dynamically typed, runs by interpreting bytecode for a register-based virtual machine, and has automatic memory management with incremental garbage collection, making it ideal for configuration, scripting, and rapid prototyping.

Lua offers good support for object-oriented programming, functional programming, and data-driven programming. Lua's main type is the table, which implements associative arrays in a very efficient manner. An associative array is an array that can be indexed not only with numbers, but also with strings or any other value of the language. Tables have no fixed size, can be resized dynamically, and can be used as "virtual tables" over another table, to simulate various object-oriented paradigms. Tables are the only data structuring mechanism in Lua[4], yet a powerful one. We use tables to represent ordinary arrays, symbol tables, sets, records, queues, and other data structures, in a simple, uniform, and efficient way. Lua uses tables to represent packages as well.

Why not Python? It is hard to talk about a language without starting a flame war. While Lua is well known in the gaming programmer community (because of its speed advantage and great embedding capabilities), Python is more popular in a more general public. With no doubt, Python ships with more libraries. However, *with no doubt*[5], "Integrating Lua with C is so easy a child could do it. Lua was designed for this also, from the beginning, and it shows[6]. This means that with a few hours' work, any C or C++ library can become a Lua library.". Another key advantage of Lua is its embedding capabilities: once code has been prototyped, it can be turned into a final system/product with very little extra work. Extra performance can be obtained using LuaJIT, yielding C-like performance for most of the pure Lua code. Lua being written in pure ANSI C, it can be easily compiled for arbitrary targets (cell-phones, embedded CPUs in FPGAs, DSP processors, ...). Adding Lua' speed advantage, the choice was a "no brainer".

Lua satisfied our initial constraints (2) and (3). In the next section, we show how we satisfied constraint (1), that is, how we made easy to develop numerical algorithms, by developing a *Tensor* object, which serves as a "glue brick" between all our library interfaces.

Torch5 is our previous version of Torch[7], and was already leveraging Lua, as we do in Torch7. In contrast, Torch7 allows the user to switch easily (on the fly) between floating types (float, doubles, or CUDA), and has parallelization (OpenMP & CUDA) capabilities.

2 Efficient N-dimensional Tensor Object

Torch7 heavily relies on its Tensor class (provided by our own standalone C Tensor library), which extends Lua's basic set of types to provide an efficient multi-dimensional array type. Most packages in Torch7, or third-party packages that depend on Torch7 rely on its Tensor class to represent signals, images, videos..., allowing us to nicely "glue" most libraries together. The Torch7 Tensor library provides a lot of classic operations (including linear algebra operations), efficiently implemented in C, leveraging SSE instructions on Intel's platforms and optionally binding linear algebra operations to existing efficient BLAS/Lapack implementations (like Intel MKL). As we will see in the next section, we also support OpenMP instructions and CUDA GPU computing.

The following code demonstrates a few standard Tensor-based operations:

```
1 t = torch.FloatTensor(100,100) -- create a tensor of single-precision floats
2 l = lab.randn(100,100)         -- randomized: sampled from a normal distribution
3 r = t + l/2                    -- basic operators
4 r:add(0.5, t)                  -- in-place operators
5 r = lab.log(lab.exp(-r)+10)    -- common operators
```

[4] Lua also allows easy interfaces with C data structures, thanks to its C API.

[5] Quoting a post on http://www.stackoverflow.com.

[6] Some might argue that SWIG (http://www.swig.org/) would ease the interface of C/C++ libraries with Python. We did use SWIG with Lua for a while, but we found it had more overhead than using the Lua C API itself, while lacking flexibility. (Try and you will see!).

[7] http://torch5.sourceforge.net.

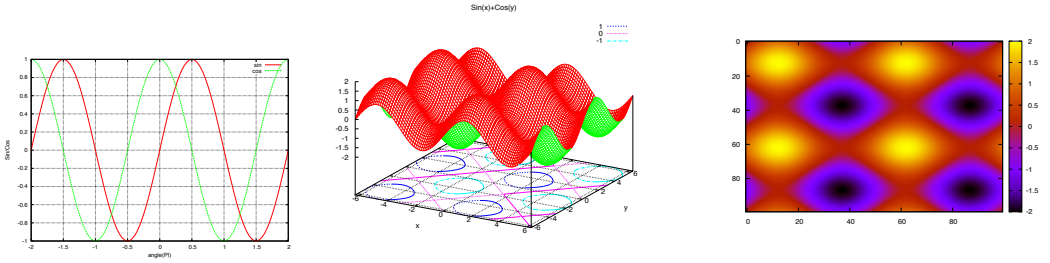

Figure 1: Sample plots produced by Torch7 plot package. Left: Simple line plot. Middle: Surface plotting using 2D tensors. Right: Matrix plotting using heat map.

As in Matlab, multiple types can co-exist in Torch7, and it is easy to go from one to the other:

```
float = torch.FloatTensor(100)  -- a single-precision tensor
double = float:double()         -- converted to double-precision
r = torch.FloatTensor(double:size())
r:copy(double)                  -- automatically casts from double->float
```

Related Approaches The Tensor idea existed long before Torch. We got mostly inspired from SN [2] & Lush [3] toolboxes, which were (to our knowledge) the first to introduce the concept (in a LISP language framework). More recently, Numpy[8] proposed the same kind of library, but for the Python language. Obviously, if we had chosen Python, we would have sticked to Numpy. As mentioned before, we nevertheless believe that Lua is a better choice for us because of the (1) speed advantage and (2) interface with any C library made really easy: in the end, our Tensor library (or Numpy) is quite small compared to all the other modules we have in Torch.

3 Torch7 Packages

At this time, Torch7 comes with 8 built-in packages:

torch: Torch7's main package: provides Tensors, easy serialization and other basic functionalities.

lab & plot: These two packages provide standard Matlab-like functions, to create, transform and plot Tensors as shown in Figure 1.

qt: Full bindings between Qt and Lua[9], with transparent conversion of Torch7 Tensors from/to QImages. Great for quickly developing interactive demos with s nice GUI (running natively on Linux, Mac or Windows platforms).

nn: The nn package provides a set of standard neural network modules, as well as a set of container modules that can be used to define arbitrary directed (acyclic or not) graphs. By explicitly describing the graph's architecture, using pluggable modules, we avoid the complexity of a graph parser, or any other middle-ware compiler.

In practice, most networks are either sequential, or have simple branching patterns and recursions. The following example shows how to describe a multi-layer perceptron:

```
mlp = nn.Sequential()
mlp:add(nn.Linear(100,1000))
mlp:add(nn.Tanh())
mlp:add(nn.Linear(1000,10))
mlp:add(nn.SoftMax())
```

Each module, or container provides standard functions to compute its output state, and back-propagate derivatives to its inputs, and to its internal parameters. Given the previous network, an input X, and the gradient of some error E with respect to the output Y—dE/dY—these three functions can be called like this:

[8] http://numpy.scipy.org.

[9] Thanks to Léon Bottou for this huge piece of work.

```
Y = mlp:forward(X)              -- compute the activations Y = f(X)
E = loss:forward(Y,T)           -- compute some loss E = l(Y,T)
dE_dY = loss:updateGradInput(Y,T)      -- compute the gradient dE/dY = dl(Y,T)/dY
dE_dX = mlp:updateGradInput(X,dE_dY)   -- back-propagate the gradients, down to dE/dX
mlp:accGradParameters(X,dE_dY)  -- compute the gradients wrt the weights: dE/dW
```

image: An image processing package. It provides all the standard image processing functions: load/save images, rescale/rotate, remap colorspaces, convolve, gaussian kernels, ...

third-party: On top of these packages, an ever growing list of third-party packages is available. Some convenient packages are: *optim*, a compact package providing steepest descent, conjugate gradient and limited memory BFGS implementations. *unsup* contains several unsupervised learning algorithms like K-means, sparse coding and auto encoders. *mattorch*, which provides a two-way interface between Matlab's matrix format and Torch's tensor; *parallel*, which provides simple routines to fork and execute Lua code on local or remote machines, and exchange data using Torch7's serialization mechanism; *camera*, a simple wrapper to camera/webcam drivers on Linux and MacOSX; *imgraph*, a package that provides lots of routines to create edge-weighted graphs on images, and manipulate these graphs. And the list never stops growing, as Lua makes it easy to interface any C library.

4 Torch7 is efficient (actually, it is the most efficient)

Torch7 has been designed with efficiency in mind, leveraging SSE when possible and supporting two ways of parallelization: OpenMP and CUDA. The Tensor library (interfaced with the "torch" package in Lua) makes a heavy usage of these techniques. From the user viewpoint, enabling CUDA and OpenMP can lead to great speedups in any "Lua" script, at zero implementation cost (because most packages rely on the Tensor library). Other packages (like the "nn" package) also leverage OpenMP and CUDA for more specific usages not covered by the Tensor library.

4.1 OpenMP support

Open Multi-Processing (OpenMP) provides a shared memory CPU parallelization framework on C/C++ and Fortran languages on almost every operating system and compiler toolset. It generally requires minimal modification for integrating into an existing project. Torch7 is designed and developed to use OpenMP directives for various operations in its tensor library and neural network package. Although the details of the OpenMP specification is beyond the scope of this work, below we show one of the most commonly used OpenMP directive, parallelization over for-loops:

```
1 #pragma omp parallel for private(i) // private makes a copy for each thread
2 for (i=0; i<N; i++)
3 {
4   a[i] = i*i;
5 }
```

Without the **omp parallel for** directive at line 1, this piece of code will run to completion on a single core. However, since each loop iteration is independent from each other, it becomes a trivial single line addition to existing code that parallelizes this computation over many cores.

Torch7 automatically detects if the compiler supports OpenMP directives and compiles a high level package that adds multi-threaded tensor operations, convolutions and several neural network classes. The switch from single threaded code to multi-threaded code is completely transparent to the user and it only requires *-l openmp* argument to be passed to executable. With this option, Torch7 by default uses the OpenMP enabled function calls when available. The number of threads to be used can be specified by either setting and environment variable to desired number:

```
bash# export OMP_NUM_THREADS=4
```

or from inside lua by

```
openmp.setNumThread()
```

function. Moreover, openmp can even be temporarily enabled or disabled using the following function calls.

```
openmp.enable()
openmp.disable()
```

Mutli-threading of BLAS operations rely on the specific BLAS library that Torch7 is linked against. For example Intel's MKL library also uses OpenMP for parallelizing Level 3 BLAS operations. In the neural network package *nn*, the convolutional layers, most common non-linearity functions like tanh and sigmoid, pooling operations like average, sum and max pooling and various other primitive operations like sum, square modules are all parallelized. For all the models that apply elementwise operations, the parallelization is almost as trivial as shown in the example above. For more complicated modules like convolutional layers with multiple input output feature maps, the function evaluation pass is parallelized over output feature maps so that every output feature is calculated in parallel. For calculating the gradient wrt kernels, operations are parallelized over kernels and over input features for gradient wrt inputs. Using this strategy the convolutional network architecture can be sped up almost linearly.

4.2 CUDA support

CUDA (Compute Unified Device Architecture) is nVidia's framework for programming their graphics processors to perform general purpose computations. CUDA exposes the hierarchy of memories available to the graphics processor, the two main ones being the external (large, high-latency) DRAM and the internal shared memory (a couple of kB, low-latency). It also exposes the hierarchy of compute cores, and how they interact with each other, and with the different types of memory.

Writing CUDA code (kernels) turned out to be simpler than expected. Contrary to common sayings, it is very easy to obtain decent performance, and the simplest kernels already yield satisfying speedups over regular C. The only three things to know, and carefully handle are: understand the interaction between shared memory and threads; understand memory coalescing, to maximize bandwidth to/from external DRAM; understand the hierarchy of processing units, to efficiently divide the workload between blocks and threads. Once understood, these concepts were sufficient to allow us to write our own 2D convolutions, which are computed at about 200GFLOP/s on a GTX580, for large enough inputs. For smaller inputs, our OpenMP+SSE implementation remains more efficient.

Once built against CUDA, Torch7 provides a new Tensor type: torch.CudaTensor. Once created, such a Tensor lives in the GPU's DRAM memory. All operators defined on standard Tensors are also defined on CudaTensors, which completely abstracts the use of the graphics processor. Here is a small illustrative example, that demonstrates the simplicity of the interface:

```
tf = torch.FloatTensor(4,100,100) -- lives in the CPU's DRAM
tc = tf:cuda()                    -- lives in the GPU's DRAM
tc:mul(3)                         -- performed by the GPU
res = tc:float()                  -- res lives in the CPU's DRAM
```

On top of the Tensors' main operators, all the matrix-based operators are available, as well as most standard convolution routines.

4.3 Benchmarks

In a recent paper [1], the authors introduced a new compiler for mathematical expressions, built upon Python and Numpy. As for Torch7, Theano is (at this time) mainly used in a neural network framework. Theano can be either run on a CPU or a GPU. The authors of Theano showed benchmarks (involving the training of various neural networks architectures) *crushing* other alternative implementations (when running Theano over a GPU), including Torch5, Matlab with GPUmat (running over a GPU) or EBLearn. We decided to reproduce these exact benchmarks, limiting ourselves to Torch7 versus Theano, as Theano appears already faster than any existing implementation.

To stay fair, we compiled both Numpy (on which Theano relies) and Torch7 against MKL Intel library. We ran the experiments on a Intel i7 950 with 4 cores. We optionally used a nVidia GTX 460 GPU. Following [1] benchmark suite, we considered the training of three kinds of multi-layer Perceptrons (with 784 inputs, 10 classes, cross-entropy cost, and respectively no-hidden layer, one hidden layer of size 500 and three hidden layers of size 1000). We also considered the training of three kinds of convolutional neural networks (on 32×32, 96×96, and 256×256 input images),

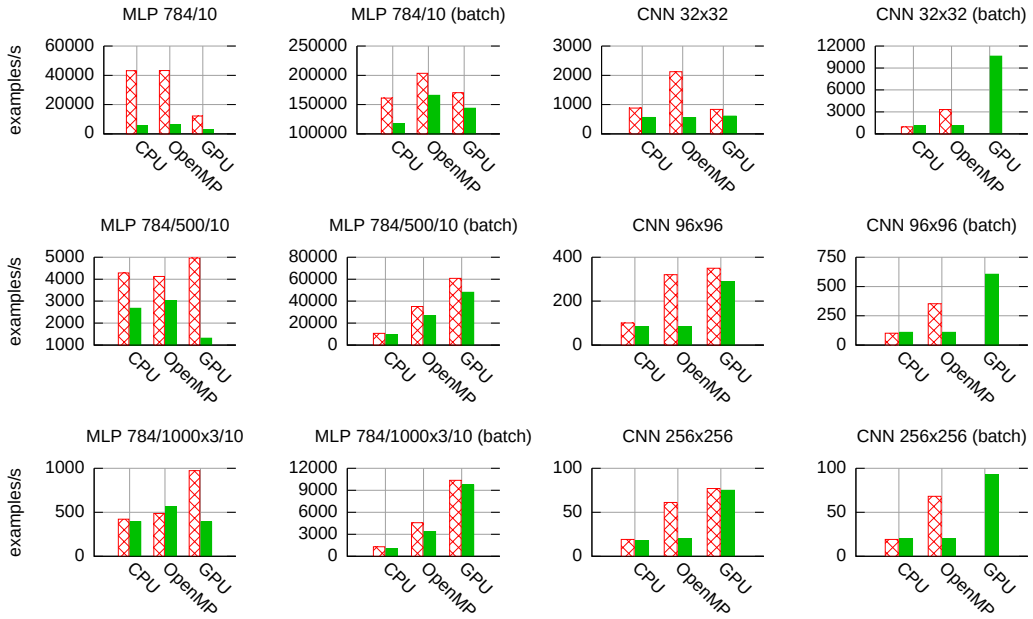

Figure 2: Benchmarks of Torch7 (red stripes) versus Theano (solid green), while training various neural networks architectures with SGD. We considered a single CPU core, OpenMP with 4 cores and GPU alternatives. Performance is given in number of examples processed by second (higher is better). "batch" means 60 examples were fed at the time when training with SGD. Note that we do not handle batch convolutions using CUDA yet (but we will in few days!).

following exactly the architectures of [1]. The optimization algorithm we used was pure stochastic gradient descent (SGD), or SGD with a mini-batch of 60 examples. We compare all architectures running on a single CPU core, over 4 cores thanks to OpenMP, or over the GPU. Note that Theano does not support OpenMP. However, it gets a speedup (on the multi-layer Perceptron benchmarks), thanks to the Intel MKL library (called through Numpy) which does supports OpenMP.

As shown in Figure 2, Torch7 is faster than Theano on most benchmarks. Interestingly, Theano really lags behind for small architectures, which might be explained by a heavy Python overhead. Another interesting comment is the great performance of OpenMP compared to the GPU implementation: only largest architectures will benefit from the GPU.

Acknowledgments

We'd like to thank our wives and families for staying nice while we were spending all our nights coding Torch7.

References

[1] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, and Y. Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, 2010. 4.3

[2] L.-Y. Bottou and Y. LeCun. Sn: A simulator for connectionist models. In *Proceedings of NeuroNimes 88*, Nimes, France, 1988. 2

[3] Y. LeCun and L. Bottou. Lush reference manual. Technical report, 2002. code available at http://lush.sourceforge.net. 2