Implementing Neural Networks Efficiently

Ronan Collobert[1], Koray Kavukcuoglu[2], and 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. Neural networks and machine learning algorithms in general require a flexible environment where new algorithm prototypes and experiments can be set up as quickly as possible with best possible computational performance. To that end, we provide a new framework called Torch7, that is especially suited to achieve both of these competing goals. *Torch7* is a versatile numeric computing framework and machine learning library that extends a very lightweight and powerful programming language Lua. Its goal is to provide a flexible environment to design, train and deploy 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 also easily be interfaced to third-party software thanks to Lua's light C interface.

Runtime efficiency is probably perceived as the most important topic when considering an efficient neural network implementation. One should however not under-estimate the time spent in designing the right neural network for a given task, or even the amount of work put into feeding data to the neural network properly. *Designing* the right network for a given task in a short amount of time requires a flexible development environment and a properly designed neural network toolbox.

Several efficient (in terms of runtime execution) neural network libraries for very specific needs are freely available. QuickNet[5] is a good example in the speech recognition community: it implements most commonly used algorithms, that is multi-layer perceptrons with few layers. However, flexible libraries are quite rare. It is not a trivial task to implement a library supporting a wide range of complex networks (such as convolutional networks for images, text or speech...), any type of connectivity (full connectivity, shared weights, order in layers...), or several type of training algorithms (stochastic gradient, batch, second order like LBFGS...). It is even more difficult to find a unified environment where one can

[5] http://www.icsi.berkeley.edu/Speech/qn.html.

easily read, prepare, feed properly the data to the network, or debug the internals of the architecture (for example when the network is not training properly).

In Section 1, we will consider efficient neural network implementation in terms of *efficient environment*. We will then focus on the *runtime efficiency* and analyze different state-of-the-art approaches to speed-up the network training and testing phases in section 2. In this work, our analysis is built on the experience we acquired with our own neural network implementation, *Torch*[6], and more particularly the last version *Torch7*.

1 Efficient Environment

An efficient environment for implementing neural networks should not be only limited to neural networks themselves. It should *provide all necessary tools for efficient development of new numerical algorithms in general*. Often one needs various numerical algorithms to transform the data before feeding it to the neural network. Algorithms will strongly vary from one research domain to the other. Moreover, in the last few years, the research activity on neural networks started to intersect with many other domains like optimization, linear algebra, parallel processing to name a few. A successful framework should provide necessary tools to cope with the variability in the development process. Only in that case the framework would allow to not only easily investigate new types of models or new training algorithms, but also to easily compare or combine neural networks with other machine learning algorithms.

In order for a framework to provide necessary environment for development of new numerical algorithms, its *extension capabilities* should be very advanced. Machine learning researchers face many problems where there is need for using existing libraries. As we will see in Section 2, this includes interfacing efficient linear algebra libraries or even the neural network implementation itself. The ability to interface these existing libraries with as little runtime and code development overhead as possible is crucial for an efficient toolbox.

Finally, the *neural network toolbox implementation itself should be modular enough* to allow for the creation of any kind of new neural network models or implementation on different modalities of data, leaving the choice of the architecture as much as possible to the user.

In this section, we will cover the following three important points: efficiency of development environment, extension capabilities and modular neural network toolbox. The modular structure of *Torch7* that fuses advantages of high-level and low-level libraries is shown in Figure 1.

1.1 Scripting Language

A scripting (or interpreted) language is the most convenient solution for fast prototyping and development of new algorithms. At the same time, it is crucial

[6] http://www.torch.ch

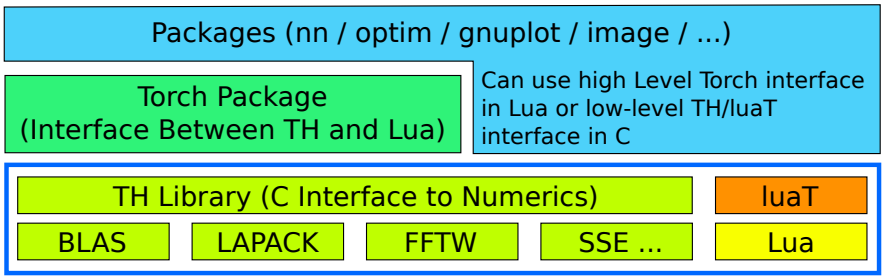

Fig. 1. Modular Structure of *Torch7*. Low level numerical libraries are interfaced with TH to provide a unified tensor library. luaT provides essential data structures for object/class manipulation in Lua. The core Torch package uses TH and luaT to provide a numerical computing environment purely in Lua. All other packages can use either Torch interface from inside Lua scripting environment or can interface low-level C interfaces for increased performance optimizations.

for the interpreted language to have a lightweight C API, both in terms of simplicity and efficiency. Simplicity in the C API encourages easier interfacing to existing external libraries and efficiency is the single most important criterion for large-scale applications.

In a complex development environment, the scripting language becomes the "glue" between heterogeneous components: different structures of the same concept (coming from different libraries) can be merged together using a high-level language, while keeping all the functionalities that are exposed from all the different libraries.

Lua Among existing scripting languages[7] finding the ones that satisfy *runtime efficiency* severely restricts the number of possibilities. In our machine learning framework *Torch7*, we chose Lua, the fastest interpreted language (with also the fastest Just In Time-JIT compiler[8]) we could find. Lua has also the advantage that it is designed to be easily *embedded* in a C application, and provides a *very clean* C API, based on a virtual stack to pass values and carry out function evaluation from C. This unifies the interface to C/C++ and minimizes the effort required for wrapping third party libraries.

Lua is intended to be used as a powerful, light-weight scripting language for any program that needs one. It is implemented as a library, written in pure C in the common subset of ANSI C and C++. Quoting Lua webpage[9],

> Lua combines simple procedural syntax with powerful data description constructs based on associative arrays and extensible semantics. Lua is

[7] E.g. on http://shootout.alioth.debian.org.
[8] http://luajit.org/
[9] http://www.lua.org.

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. As shown in Figure 2, it handles numerical computations very efficiently (compared to C). This is a great asset for rapid implementation of new numerical algorithms. Lua's main type is table, which implements associative arrays in a very efficient manner (see Figure 2). 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, yet a powerful one. One can 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. In addition, functions are first class citizens of the language. A function, just like any other variable can be passed as a variable to or returned from a function. Last, but not the least, Lua supports closures. Combined with tables, closures provide a very powerful and efficient syntax for data handling and programming complicated algorithms.

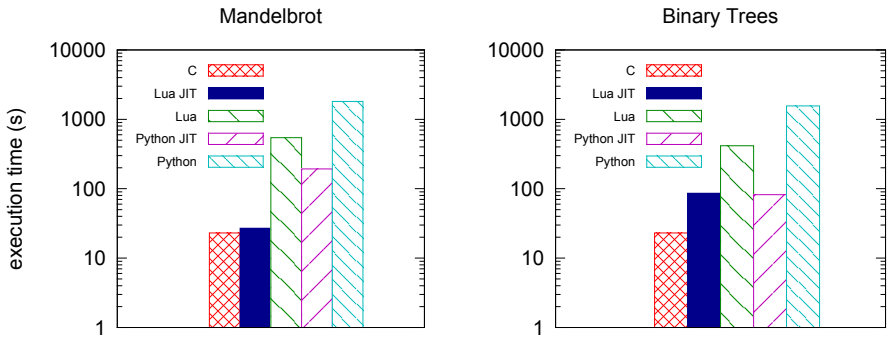

Fig. 2. Comparison of runtime efficiency of the C language (with gcc 4.4), Lua 5.1.4 and Python 2.7. Lua and Python JIT implementations were LuaJIT and PyPy, respectively. The Mandelbrot and Binary Trees benchmarks are taken from "The Computer Language Benchmarks Game". All benchmarks were run using a single CPU on a high-end 12 cores Xeon server. The Mandelbrot benchmark makes a heavy use of numbers, while the Binary Trees benchmark makes a heavy use of data structures (struct in C, tables in Python and Lua). The execution time is reported (on a log-scale axis) for each language.

Why not Python? It is hard to talk about a programming language without starting a flame war. While Lua is well known in the gaming programmer community, mostly due to its speed advantage and great embedding capabilities, Python is more popular in more general public. With no doubt, Python ships with more libraries and one can find support about almost any problem easily in many different contexts. However, Lua offers at least two important advantages over Python:

- First and foremost, the simplicity of integrating existing C/C++ libraries is very important. Many efficient numerical algorithms are implemented in specialized packages in BLAS, LAPACK, FFTW and similar libraries. A lightweight interface to existing code is crucial for achieving a high performance environment. In section 2.8 we show quantitative results on efficiency of Lua compared to Python when wrapping BLAS function calls.
- Second, since Lua is embeddable in C/C++, any prototyped application can be turned into a final system/product with very little extra effort. Since Lua is written in pure C and does not have dependency to any external library, it can be easily used in embedded applications like, Android, iOS [10], FPGAs [11] and DSPs.

There are also alternatives to writing a custom interface between interpreted language and C/C++, like Simplified Wrapper and Interface Generator (SWIG) [12]. Although these might provide a simplified interface at first, writing a tensor library with several linear algebra backends requires a very fine-grained control and we found it is harder to manage this interface rather than using the native Lua API.

In addition to its performance advantage on number of operations (see Figure 2), Lua also provides other unique advantages – rarely found simultaneously in existing programming languages – for implementing a large-scale machine learning framework. In the following section we will show how we extended Lua's basic numerical capabilities to a rather complete framework for developing complex numerical algorithms.

1.2 Multi-purpose Efficient N-dimensional Tensor Object

Torch7 provides a generic Tensor library (called TH) that is written in pure C. This library is interfaced in Lua, providing new efficient multi-dimensional array types in the scripting language. Most packages in *Torch7* (or third-party packages that depend on *Torch7*) rely on this Tensor class to represent signals, images, videos..., allowing Lua to nicely "glue" most libraries together. Fast prototyping and creation of new packages is made possible, as the library is available directly from both Lua and C. Interfacing or extending existing libraries is very efficient. The following code demonstrates a few standard Tensor-based operations, from the Lua side:

[10] https://github.com/clementfarabet/torch-ios
[11] http://www.neuflow.org
[12] www.swig.org

```
-- create a tensor of single-precision floats
t = torch.FloatTensor(100,100)

-- randomized: sampled from a normal distribution
l = torch.randn(100,100)

-- basic operators
r = t + l/2

-- in-place operators
r:add(0.5, t)

-- common operators
r = torch.log(torch.exp(-r)+10)
```

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

```
-- a single-precision tensor
tfloat = torch.FloatTensor(100)

-- converted to double-precision
tdouble = tfloat:double()

r = torch.FloatTensor(tdouble:size())

-- automatically casts from double->float
r:copy(tdouble)
```

A sample matrix, matrix multiplication operation is done as in the following example.

```
x = torch.Tensor(1000,5000)
y = torch.Tensor(5000,3000)
z = torch.mm(x,y)
print(z:size())

 1000
 3000
[torch.LongStorage of size 2]
```

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, OpenBLAS or ATLAS). As we will see in the next section, we also support OpenMP instructions and CUDA GPU computing for certain subset of operations where these platforms offer unique performance advantages.

Related Approaches Our Tensor library implementation got mostly inspired from SN [3] and Lush [5] toolboxes, which were one of the first to introduce

the concept (in a LISP language framework). Matlab also supports N-dimension arrays (even though early releases only supported 2D matrices). Compared to Matlab, we put big emphasis on *memory allocation control*, as we will see in Section 2.2. Numpy[13] is another popular alternative, but only available for the Python language. As mentioned before, Lua offers unique advantages for a machine learning framework because of its speed and the simpler C interface.

1.3 Modular Neural Networks

Following [6], we view a neural network as a set of modules connected together in a particular graph. In *Torch7*, 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:

```
-- compute the activations Y = f(X)
Y = mlp:updateOutput(X)

-- compute some loss E = l(Y,T)
E = loss:updateOutput(Y,T)

-- compute the gradient dE/dY = dl(Y,T)/dY
dE_dY = loss:updateGradInput(Y,T)

-- back-propagate the gradients, down to dE/dX
dE_dX = mlp:updateGradInput(X,dE_dY)

-- compute the gradients wrt the weights: dE/dW
mlp:accGradParameters(X,dE_dY)
```

The "nn" package in *Torch7* provides about 80 different neural network modules, allowing the user to implement most existing neural networks with minimal effort in pure Lua.

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

Leveraging the TH library Neural network modules in *Torch7* use Tensors provided by the TH library (see Section 1.2) to represent their own input data, output or internal states. Most modules are simply written in Lua, using the Torch package for intensive numerical operations. Only packages which require very specific operations have a dedicated C back-end. And, even in this case many of them use the TH library interface from C. In any case, Tensors are used as data containers to interact seamlessly with the rest of the library.

Training Algorithms In *Torch7*, every neural network module, given the partial derivatives with respect to its outputs, is able to compute the partial derivative with respect to its parameters and its inputs. Thus, any complicated network structure can be trained using gradient-based optimization methods. Batch, mini-batch and stochastic gradient descent algorithms are supported. More advanced algorithms, such as second-order gradient descent algorithms like conjugate gradient or LBFGS are also possible, thanks to a numerical package called "optim". While this optimization package is designed to be used stand-alone, it also provides second-order optimization capabilities for neural networks when used with the "nn" package.

1.4 Additional Torch7 Packages

Torch7 comes with many built-in and third-party packages. In order to encourage collaborations and redistribution of machine learning algorithms, a built-in package manager is provided. It can easily download, compile and install additional *Torch7* packages from any package repository, when needed. At this time, the most interesting packages related to numerical computation or numerical analysis are:

- **torch**: *Torch7*'s main package: provides Tensors, easy serialization and other basic functionalities. This package provides, Matlab-like functions to create, transform and use Tensors.
- **gnuplot**: This package provides plotting interface to Gnuplot using Tensors.
- **image**: An image processing package. It provides all the standard image processing functions such as loading and saving images, rescaling, rotating, converting color spaces, filtering operations, ...
- **optim**: A compact package providing steepest descent, conjugate gradient and limited memory BFGS implementations.
- **qt**: Full bindings between Qt and Lua[14], with transparent conversion of *Torch7* Tensors from/to QImages. Great for quickly developing interactive demos with advanced GUIs (running natively on Linux, Mac or Windows platforms).

The list of available packages is constantly growing, as Lua makes it easy to interface any C library. Third-party packages include: *unsup*, which contains

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

several unsupervised learning algorithms like 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.

2 Efficient Runtime Execution

Torch7 has been designed with efficiency in mind, leveraging SSE when possible and supporting two ways of parallelization: OpenMP and CUDA. The Tensor library (which is interfaced to Lua using the "torch" package) makes heavy use 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. In the following we explain specific advantages of *Torch7* for achieving an excellent runtime performance.

2.1 Float or Double Representations

One of the major computational bottlenecks of modern computers is their memory bandwidth. When implementing any numerical algorithm, the number of memory accesses should be always reduced by all means. This has an impact not only on the coding style, but also on the floating point type we will choose when implementing neural networks. A C "double" takes usually 8 bytes in memory, while C "float" takes only 4. Given that high precision is rarely required in neural networks, one might consider using floating point precision in most cases. On a simple matrix-matrix operation, speedups of $\times 2$ are common when using floats instead of doubles. In practice similar speedups are also observed in neural networks using floating point precision. In that respect, in *Torch7*, the user can easily choose (at runtime) the default floating point type.

2.2 Memory Allocation Control

One of the main complaints about using high level interfaces (such as Matlab) for numerical programming is the loss of control over memory allocation. The high-level abstraction makes it very hard for the researcher to know when a copy of a tensor is created. Although this is not a major problem for small-scale applications, as the data size grows, repeated copy and memory allocation might become problematic and even a bottleneck for the algorithm. To avoid such problems, *Torch7* tensor library is designed to support complete control over new memory allocation only when the user wants to use it. To better demonstrate this point, we repeat the matrix multiplication example.

```
  x = torch.Tensor(1000,5000)
  y = torch.Tensor(5000,3000)
  z = torch.mm(x,y) print(z:size())

 1000
 3000
[torch.LongStorage of size 2]
```

One can see that the tensor z, which did not exist before, is newly allocated in this context. One can imagine that these series of operations are done repeatedly inside a loop. In this case, *Torch7* allows the following intuitive syntax.

```
x = torch.Tensor(1000,5000)
y = torch.Tensor(5000,3000)
z = torch.Tensor(1000,3000)
torch.mm(z,x,y)
```

As it can be seen from the example, the *torch.mm* function also can take three arguments, in which case the first argument becomes the result of the operation. This syntax is implemented for all operations in the Tensor library consistently, so that for every single operation, the user has the choice of passing in the target Tensor or allocating a new one without any overhead and heavy syntax. For example the following element-wise *Sin* operation can be represented in two different ways.

```
  x = torch.rand(1000)

  -- a new tensor is created
  tsin = torch.sin(x)

  -- a scalar one is added to tensor x (x is reused)
  x:add(1)

  -- tsin is reused
  torch.sin(tsin,x)
```

In this example, both scalar addition to tensor x and calculating the *Sin* of resulting tensor did not allocate any new memory. In the above example, we also hinted another use of tensor library, where one can make method calls on a tensor object, as in any object oriented language. This syntax makes it explicit that the operation is directly applied on the object that the method call is done.

2.3 BLAS/LAPACK Interfaces

The key to a successful numerical computation framework is to have efficient implementations of linear algebra operations. This requires highly sophisticated algorithms with very precise implementations. In order to be able to provide the best experience, the C tensor library (TH) that is included in *Torch7* interfaces BLAS and LAPACK libraries.[15] All the major Level 1, 2 and 3 BLAS operations like matrix-vector products, matrix-matrix products and most major linear algebra routines like singular value

[15] http://www.netlib.org.

decomposition, matrix inverse, least square solutions are interfaced to BLAS and LAPACK libraries respectively. This interface provides the user with a rich experience of building block operations where higher level algorithms can easily be implemented.

2.4 SIMD Instructions

Most computations involved in a neural network consist in applying the same type of operations over (possibly large) vectors or matrices. Several CPU architectures, such as PowerPC, Intel or ARM support SIMD (Single Instruction, Multiple Data) operations which are perfectly suited for this kind of task: for example with SSE (Streaming SIMD Extensions) on Intel processors one might perform 4 additions over a vector with a unique instruction. Calling these instructions instead of regular CPU instructions might lead to great speedup. This type of optimization is unfortunately CPU-specific. Fortunately, in many cases one can rely on BLAS/LAPACK implementations specialized for a given platform, which leverage SIMD instructions. For other neural network specific cases, such as convolutions, one must implement specialized routines for each architecture of choice. In *Torch7*, we try to leverage SSE (on Intel architectures) and NEON (on ARM architectures) whenever possible. Compared to a non-SSE implementation 1.5× speedup are common, as shown in Figure 3 in the case of convolutional neural networks.

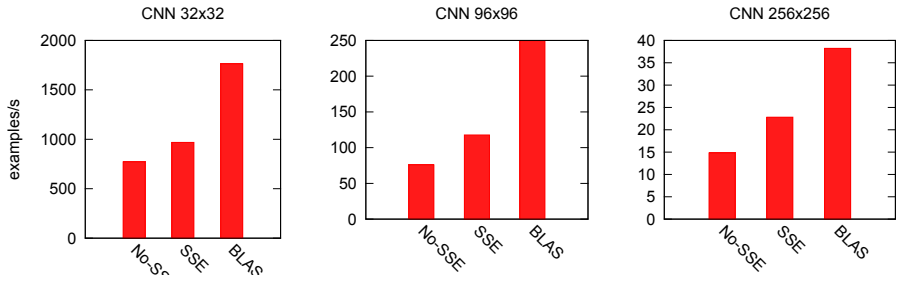

Fig. 3. Comparison of several convolutional neural network implementations (without SSE, with SSE or with BLAS). Tests were conducted using one core on a Intel bi-Xeon X5690 server. Performance is given in number of examples processed by second (higher is better). Three different architectures were tested, with input image sizes of 32x32, 96x96 and 256x256 respectively.

2.5 Ordering Memory Accesses

As already mentioned in Section 2.1 and Section 2.2, memory accesses are one of the main bottleneck on today's computers. In fact, not only the *number* of accesses is important, but also the *order* of these accesses. For example, operations with tensors not contiguous in memory (say, with extra jumps between each element of the tensor) should always be avoided. In many cases, it is better to organize the tensor in a contiguous memory block (possibly at the cost of the copy), before performing any intensive

computations. A striking example for neural networks is convolutions. When performing a convolution over an image, successive dot products are done between the kernel and all possible patches of the image. One can create a copy of all these patches beforehand (the drawback being a huge memory cost for large convolutions or large images) and then apply a matrix-matrix operation (using BLAS) to compute all dot products. The memory consumption increases proportional to the number of pixels of convolutional kernel. As shown in Figure 3, this leads to unbeatable runtime performance, even though the initial memory copy is quite large. *Torch7* provides this implementation as part of the neural net package too. Whenever there is sufficient memory available, it is advantageous to use this implementation which uses an innovative design that takes advantage of multi-core CPU architectures.

2.6 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   // private makes a copy for each thread
2   #pragma omp parallel for private(i)
3   for (i=0; i<N; i++)
4   {
5     a[i] = i*i;
6   }
```

Without the **omp parallel for** directive at line 2, this piece of code will run to completion using a single thread. 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 torch 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 the "OMP_NUM_THREADS" environment variable to desired number:

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

or from inside lua by

```
1  torch.setnumthreads(4)
```

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

```
1  torch.setnumthreads(1)
2  torch.setnumthreads(N)
```

Multi-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 element-wise 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 feature maps for gradient wrt inputs. Using this strategy the convolutional network architecture can be sped up almost linearly.

2.7 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.

Contrary to common belief, we found that writing CUDA code (kernels) can be significantly simplified. 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: understanding 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. It is worth mentioning that *Torch7* employs an efficient, yet general method for implementing a wide variety of CUDA kernels. As it is shown in the upcoming sections, this strategy results in the best performance in most cases. However, it is also possible to achieve superior performance by developing CUDA kernels under specific assumptions, like particular input or operator shape and sizes. Despite the performance advantage, these cases generally require significant development effort and produce modules that can not be reused, thus they are not suitable for a general machine learning library.

Once built with CUDA, *Torch7* provides a new Tensor type: torch.CudaTensor. Tensors with this particular type 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:

```
1   -- lives in the CPU's DRAM
2   tf = torch.FloatTensor(4,100,100)
3
4   -- lives in the GPU's DRAM
5   tc = tf:cuda()
6
```

```
7    -- performed by the GPU
8    tc:mul(3)
9
10   -- res lives in the CPU's DRAM
11   res = tc:float()
```

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

2.8 Benchmarks

In this section we analyze the efficiency of *Torch7* in two different setups: first in the framework of a matrix-matrix multiplication benchmark, then when training various neural networks. To that effect, we compare *Torch7* with *Numpy* and *Theano*. We chose *Numpy* as a reference because it is a widely-used numerical library for Python, the latter being itself a widely-used scripting language. Theano [1] is a recent compiler for mathematical expressions, built upon Python and Numpy, and which has been shown as over-performing many neural network implementations, which makes it a very relevant baseline. In our experiments we chose the latest version of each software, that is *Theano* 0.5, *Numpy* 1.6.1 and *Scipy* 0.10.1.

Measuring the Overhead of Interpreted Languages The majority of the computation for neural networks and many numerical algorithms is spent in BLAS calls for performing linear algebra operations. To that end, we demonstrate the efficiency of the *Torch7* and also the underlying C library TH.

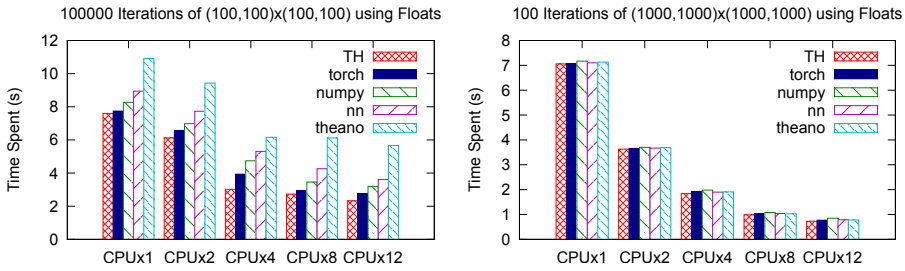

Fig. 4. Benchmarks of matrix multiplication performance using C, *Torch7* torch package, nn package in *Torch7*, Numpy and Theano. Tests were conducted on a machine with two Intel Xeon X5690 CPUs with 6 computational cores in each CPU. Hyper-threading was disabled. We considered multi-thread computation using 1, 2, 4, 8 and 12 CPU cores. Performance is given in seconds of time spent for processing, therefore smaller is better.

The *Torch7* numerical routines follow a simple design that contains layers. The first layer is an efficient C library that provides a high level tensor package (TH). TH library provides a templated design that enables the choice of different precisions.

Available types are, Byte (unsigned char), Char (char), Short (16 bit integer), Integer (32 bit integer), Long (64 bit integer), Float (32 bit floating point) and Double (64 bit floating precision). TH library does not have any dependencies to Lua or any other language other than standard C libraries, therefore it is also suitable to be used by other projects that rely on efficient numerical routines. The choice of C language was a careful choice as with Lua. Since TH uses only C, it can be compiled in almost any programming environment like cellphones, DSP, embedded systems, etc. TH provides interface to many BLAS operations, but also contains hand-coded operations for all functions in case no BLAS library is available. It also provides and interface to several most widely used LAPACK routines for linear algebra. The second layer on top of TH is the *torch* package that integrates TH into Lua. All of the TH mathematical operations are interfaced from the Lua language in the torch packages. Finally, the *nn* package uses the *torch* package to provide a modular, yet fast and efficient, neural network library.

One might argue that such a layered approach would introduce quite a bit of overhead. In order to quantify the overhead coming from each layer, we selected matrix-matrix multiplication as our test case since it is one of the most widely used operations in linear algebra and ran tests using different sizes of matrices and different layers of programming. We used 100×100 and 1000×1000, matrices and benchmarked using a C only program that directly uses TH library, using torch library, using linear layer (with no bias) from *nn* package in Torch7. We also included tests using *Numpy* package and finally *Theano*. We compiled all packages using Intel MKL library to be able achieve the best possible performance and maximize the advantages of using CPU threading. As it can be seen from the results given in Figure 4, the overhead coming from TH, *Torch7* or *nn* libraries is minimal, even for small size matrices. Even though Python gets a bit more overhead, for larger matrices the overhead is minimal in all configurations.

Comparing Machine Learning Packages 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) comparing with other alternative implementations (when running Theano over a GPU), including Torch5, Matlab with GPUmat (running over a GPU) or EBLearn [16]. Below, we reproduce these exact benchmarks, limiting ourselves to *Torch7* versus Theano, as Theano appears already faster than any existing implementation.

For a fair comparison, we compiled both Numpy and SciPy (on which Theano relies) and *Torch7* against MKL Intel library. Latest versions of Theano also support direct link against MKL for certain operations (without passing by Numpy), which we setup carefully. We ran the experiments on a Intel Xeon X5690 with 12 cores. We optionally used a nVidia Tesla M2090 GPU. Following [1] benchmark suite, we considered the training of three kinds of multi-layer Perceptrons. **1.** 784 inputs, 10 classes, cross-entropy cost, and respectively no-hidden layer. **2.** One hidden layer of size 500. **3.** Three hidden layers of size 1000. We also considered the training of three kinds of convolutional neural networks (as shown in Table 1) on 32×32, 96×96, and 256×256 input images, following exactly the architectures given in [1]. The optimization algorithms we used were pure stochastic gradient descent (SGD) and SGD with a mini-batch of 60 examples. We compare all architectures running on a single CPU core, over

[16] http://www.eblearn.sf.net

Table 1. Convolutional Network Architectures used in the benchmark study.

	32×32	96×96	256×256	# F. Maps
1.c Convolution	5×5	7×7	7×7	6
1.p Max-pooling	2×2	3×3	5×5	6
2.c Convolution	5×5	7×7	7×7	16
2.p Max-pooling	2×2	3×3	4×4	16
3.l Linear	120 output features			
4.o Linear	10 output features			

multiple cores using OpenMP, or on the GPU. Note that Theano does not support OpenMP. However, it gets a speedup (on the multi-layer Perceptron benchmarks), since the Intel MKL library (called through Numpy) supports multiple threads using OpenMP.

As shown in Figure 5, *Torch7* is faster than Theano on most benchmarks. Interestingly, Theano underperforms for small architectures using pure SGD training (left column in Figure 5), which might be explained by a Python overhead, as mentioned in the previous section. Another interesting comment is the surprising performance of OpenMP implementations compared to the GPU implementation. As it can be seen from the graphs only largest network architectures will benefit from using the GPU. It is also worth mentioning that for CNN with 32×32 inputs using batch training, Theano's GPU implementation is superior than *Torch 7*. Under certain conditions, GPU optimizations might pay off by providing significant speed-ups, however they also require significant development effort for covering a small input domain. For CNN experiments a second *Torch7* benchmark, TorchMM is included. In this case matrix-matrix product operations for performing convolutions as explained in section 2.5 are used. It can be seen that this implementation significantly outperforms other models from *Theano* and *Torch7*, including GPU implementations.

3 Efficient Optimization Heuristics

As pointed in Chapter **LeonBottou**, the size of datasets have grown faster than the speed of processors in the last couple of years. When estimating the parameters of a neural network, it is then crucial to use an optimization procedure that can scale accordingly. Recently, research on optimization methods for neural networks has become an important topic [2,4,8,7]. *Torch 7* provides a flexible framework designed particularly to make it easy for developing optimization algorithms on neural networks.

Let us consider the case of supervised learning, when one has a training set of N examples (x_n, y_n), with x_n an observed input vector and y_n an output target vector that we wish to predict. We consider a loss function $l(\hat{y}_n, y_n)$ that measures the cost of predicting $\hat{y}_n$ when the actual answer is y_n. We also consider a predictor $f_{\mathbf{w}}(x_n)$, with trainable parameters $\mathbf{w}$. The task of learning can be defined as finding the vector $\mathbf{w}$ that minimizes the loss function L over the entire training set:

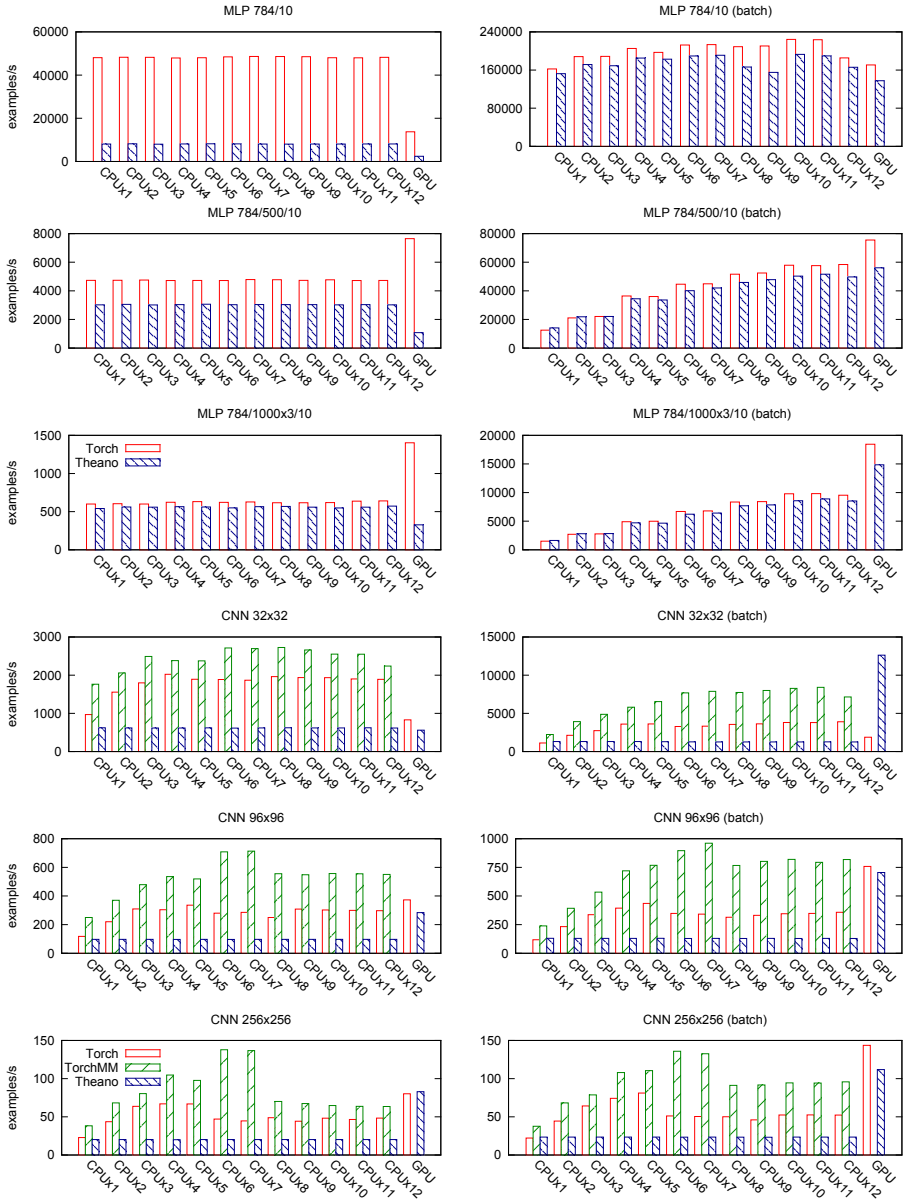

Fig. 5. Benchmarks of *Torch7* versus Theano, while training various neural networks architectures with SGD algorithm. Tests were conducted on a machine with two Intel Xeon X5690 CPUs and Nvidia M2090 GPU. We considered multi-thread computation using 1 to 12 CPU cores using OpenMP and GPU with Nvidia CUDA interface. Performance is given in number of examples processed by second (higher is better). "batch" means 60 examples at a time were fed when training with SGD. *TorchMM* uses the convolutional neural network layer implementation introduced in Section 2.5.

$$L(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N} l(f_{\mathbf{w}}(x_n), y_n), \tag{1}$$

$$\mathbf{w}^* = \operatorname*{argmin}_{\mathbf{w}} L(\mathbf{w}). \tag{2}$$

This general form of loss minimization can be easily carried out using one of a variety of optimization methods like Conjugate Gradient Descent (CG), BFGS or Limited Memory BFGS, Levenberg-Marquardt methods or simple SGD. In Torch7, these heuristics and methods can be carried out simply, using one unifying idea: decoupling the form of the function $f_{\mathbf{w}}$ from the optimization procedure. By grouping all the trainable parameters into a single parameter vector and using a vector of gradients of the same size for gradients, the type and shape of the neural network is completely abstracted from the developer. Combined with powerful closure mechanism of Lua, one can develop optimization algorithms for most complicated neural network models as easy for the simplest ones. The following code shows how this is done:

```
1   -- create an arbitrary model:
2   model = nn.Sequential()
3   model:add( nn.Linear(100,1000) )
4   model:add( nn.Tanh() )
5   model:add( nn.Linear(1000,10) )
6
7   -- and a loss function:
8   loss = nn.MSECriterion()
9
10  -- extract the parameters, and the gradient holder
11  w,dloss_dw = model:getParameters()
12
13  -- w and dl_dw are two vectors of the same size
```

Once the trainable parameter vector has been extracted, arbitrary, external optimization procedures can be used. *Torch7* provides a few standard methods (LBFGS, CG, SGD, ASGD) which simply require: (1) a function that computes L_w and $\frac{dL}{dw}$ and (2) the parameter vectors $\mathbf{w}$ and $\mathbf{dL/dw}$. Of course, L_w can be either the true loss, or any approximation of it. The function that is defined is responsible for sampling from the training dataset, and estimating these approximations.

With these two concepts in mind, one can easily define a loop over a training dataset, and define a closure at each iteration, which computes L_w and $\frac{dL}{dw}$. The following listing shows an example of such a loop, assuming a pre-shuffled training dataset in which each entry is a tuple (x_n, y_n):

```
1  -- assuming a training dataset 'trainset', and the model
2  -- defined above: 'model', 'w' and 'dL_dw':
3  for e = 1,nepochs do
4     for i,sample in ipairs(trainset) do
5        -- next training pair:
6        x_n = sample[1]
7        y_n = sample[2]
8
9        -- create closure that estimates y_n_hat = f_w(x_n),
```

```
10        -- stochastically
11        feval = function()
12           -- estimate loss:
13           y_n_hat = model:forward(x_n)
14           f = loss:forward(y_n_hat, y_n)
15
16           -- estimate gradients:
17           dloss_dw:zero()
18           dloss_dy_n_hat = loss:backward(y_n_hat, y_n)
19           model:backward(x_n, dloss_dy_n_hat)
20
21           -- return loss, and gradients
22           return f,dloss_dw
23        end
24
25        -- now that the closure is defined, pass it to an
26        -- optimization algorithm:
27        w,fs = optim.sgd(feval,w)
28
29        -- + the new w is returned, but as computations are
30        --   done in place, it is typically not necessary to
31        --   store it (the old w contains the new value)
32        -- + fs is a list of all the function (loss)
33        --   evaluations that were done during optimization.
34        --   SGD only returns one value, as it does not
35        --   perform any line search.
36     end
```

In the listing above, one can see that the loss and gradient estimation can be easily changed at runtime, and estimated over arbitrary batch sizes. To use a batch size different than 1 (as done above), one simply needs to create a list of training pairs, and the *feval* function needs to loop over these training pairs to estimate the approximate loss and gradients.

4 Conclusion

Compared to the early days of neural network training, the challenges towards an efficient implementation did not change a lot, however the means changed slightly. Already in the late 80's, the SN [3] toolbox was providing a scripting language (LISP) for building neural networks in a modular way. At the time, memory bandwidth and processor speed were about the same order of magnitude. Nowadays, we have to pay much more attention on memory accesses, counting number of instructions for optimizing the code is not sufficient anymore. Specific vectorized instructions can be easily integrated, but will not give order of magnitude speedups. In the end, what brings most advantage is parallelization. As computers become more and more parallel, it becomes crucial to leverage parallelization frameworks properly, such as OpenMP. On a more extreme side, GPUs (for e.g. running with CUDA) are not as attractive as some could have expected: GPU-specific implementations require heavy extra work for a speedup (see Figure 5) which can be quite disappointing compared to what one can get with few extra lines of code with OpenMP.

Acknowledgments

We would like to thank Léon Bottou for providing the Qt library interface and Yann LeCun and Léon Bottou for sharing their work in SN, Lush, and extending their support and advices. We would also like to thank James Bergstra for making his benchmark code available[17].

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. 2.8, 2.8, 2.8
2. L. Bottou. Large-scale machine learning with stochastic gradient descent. In Y. Lechevallier and G. Saporta, editors, *Proceedings of the 19th International Conference on Computational Statistics (COMPSTAT'2010)*, pages 177–187, Paris, France, August 2010. Springer. 3
3. L. Bottou and Y. LeCun. SN: A simulator for connectionist models. In *Proceedings of NeuroNimes 88*, Nimes, France, 1988. 1.2, 4
4. Q.V. Le, A. Coates, B. Prochnow, and A.Y. Ng. On optimization methods for deep learning. *Learning*, pages 265–272, 2011. 3
5. Y. LeCun and L. Bottou. Lush reference manual. Technical report, 2002. code available at http://lush.sourceforge.net. 1.2
6. Y. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller. Efficient backprop. In G.B. Orr and K.-R. Müller, editors, *Neural Networks: Tricks of the Trade*, pages 9–50. Springer, 1998. 1.3
7. J. Martens. Deep learning via hessian-free optimization. In *Proceedings of the 27th International Conference on Machine Learning (ICML)*, volume 951, page 2010, 2010. 3
8. O. Vinyals and D. Povey. Krylov subspace descent for deep learning. *Arxiv preprint arXiv:1111.4259*, 2011. 3

[17] http://www.github.com/jaberg/DeepLearningBenchmarks.