IDIAP RESEARCH REPORT

Torch: a modular machine learning software library

Ronan Collobert * Samy Bengio *
Johnny Mariéthoz *

IDIAP–RR 02-46

October 30, 2002

Dalle Molle Institute
for Perceptual Artificial
Intelligence • P.O.Box 592 •
Martigny • Valais • Switzerland

phone +41 – 27 – 721 77 11
fax +41 – 27 – 721 77 12
e-mail `secretariat@idiap.ch`
internet `http://www.idiap.ch`

[1] IDIAP, CP 592, 1920 Martigny, Switzerland, `{collober,bengio,marietho}@idiap.ch`

IDIAP Research Report 02-46

Torch: a modular machine learning software library

Ronan Collobert Samy Bengio Johnny Mariéthoz

October 30, 2002

Abstract. Many scientific communities have expressed a growing interest in machine learning algorithms recently, mainly due to the generally good results they provide, compared to traditional statistical or AI approaches. However, these machine learning algorithms are often complex to implement and to use properly and efficiently. We thus present in this paper a new machine learning software library in which most state-of-the-art algorithms have already been implemented and are available in a unified framework, in order for scientists to be able to use them, compare them, and even extend them. More interestingly, this library is freely available under a BSD license and can be retrieved on the web by everyone.

1 Introduction

Statistical machine learning algorithms [1, 14] can be used to construct systems able to learn to solve tasks given both a set of examples of that task which were drawn from an unknown probability distribution, and with some *a priori* knowledge of the task. On top of providing such algorithms, researchers in this field also provide means of measuring the expected performance of such systems when used on new examples drawn from the same probability distribution.

Examples of such algorithms often used by the ICASSP community range from multi-layer perceptrons and support vector machines to Gaussian mixture models and hidden Markov models. They are for instance used for signal processing (analysis and prediction), image and video processing (face, gesture or handwritten recognition) or speech processing (speech recognition or speaker verification).

While many new algorithms are proposed every year in various international conferences and journals, it is often difficult for scientists interested in solving a particular task (say speech recognition) to implement them and compare them with their usual tools.

The aim of this paper is to present **Torch**, a new machine learning software library[1] available to the scientific community under a free BSD license, and which implements most state-of-the-art machine learning algorithms in a unified framework. The objective is to ease the comparison between algorithms, and simplify the process of extending them or even adding new ones. The paper is organized as follows: in the next section, we present the main concepts of **Torch**; section 3 presents the most popular machine learning algorithms already available in the current library; section 4 compares **Torch** with other available tools; this is followed by a quick conclusion.

2 Main Concepts of Torch

Torch has been developed using an object-oriented paradigm and implemented in `C++`. In order to simplify the modification of existing algorithms or the design of new algorithms or methods, a modular strategy was chosen through the definition of the following broad classes:

- `DataSet`: this class handles data. Several subclasses provide ways to handle static or dynamic data, data that can fit into memory or which could be accessed "on-the-fly" from disk (for very large data sets for example), *etc...*

- `Machine`: this class represents any black-box that, given an (optional) input (again, either static or dynamic) and some (optional) parameters, returns an output. It could be for instance a neural network, a support vector machine, a hidden Markov model, *etc...*

- `Trainer`: this class is used to select an optimal set of parameters of a machine according to a given criterion and a given `DataSet`, and test it using another (or the same) `DataSet`.

- `Measurer`: objects of this class print in different files various measures of interest. It could be for example the classification error, the mean-squared error or the log-likelihood.

Thus, the *general idea* of **Torch** is very simple: first the `DataSet` produces one or several "training examples". The `Trainer` gives them to the `Machine` which computes an output, which is used by the `Trainer` to tune the parameters of the `Machine`. During this process, one or more `Measurer`(s) can be used to monitor the performance of the system. Note however that some machines can only be trained by specific trainers:

- various "gradient machines" (including multi-layer perceptrons) can be trained by gradient descent,

- support vector machines, for classification or regression can be trained by a trainer specialized on constrained quadratic problems,

[1] **Torch** is available at `http://www.torch.ch`.

- distributions (such as Gaussian mixture models or hidden Markov models) are usually trained using an Expectation-Maximization (or its Viterbi approximation) trainer but can also be trained by gradient descent.

3 Examples of commonly used machines

3.1 Gradient Machines

An important technique in machine learning was introduced by the back-propagation (BP) algorithm [9]. In fact, BP is the application of a simple gradient descent to complex but derivable functions. A 'gradient machine" in **Torch** corresponds to a function which can be trained by gradient descent.

More formally, suppose that we have a function $f_w(x)$ where $x \in \mathbb{R}^d$ is an input vector, and $w \in \mathbb{R}^m$ are called "the weights", derivable with respect to w. Then, given a training set $(x_i, y_i)_{i=1...T}$ ($x_i \in \mathbb{R}^d$ are the inputs, and $y_i \in R^n$ are the targets), and given a cost function $C(f(x), y)$, we would like to minimize

$$\sum_{i=1}^{T} C(f_w(x_i), y_i)$$

with respect to w. To achieve this, we often use a *stochastic* gradient descent technique as suggested in [11]: the idea is to compute for each example (x_i, y_i) the derivative of the cost function with respect to w and update weights according to the following formula:

$$w \longleftarrow w - \lambda \frac{\partial C(f_w(x_i), y_i)}{\partial w}$$

where the "learning rate" $\lambda \in \mathbb{R}$ is given by the user.

Since one can think of many useful kinds of gradient machines and cost functions, we decided to implement them in **Torch** in a very *modular* way, following an idea proposed in [2]: first, several simple "modules" can be plugged with each other in order to obtain the function $f_w(x)$ that is needed. Moreover, several cost functions C, can be chosen independently.

Let us propose an example: suppose you want to create a multi-layer perceptron (MLP), with one non-linear (say a hyperbolic tangent) hidden layer unit, and with one linear output. The function $f_w(x)$ can be written as:

$$f_w(x) = v_0 + \sum_{j=1}^{N} v_j \tanh\left(u_{j0} + \sum_{i=1}^{d} u_{ji} x^i\right)$$

where N is the number of hidden units, $w = (v_j, \ldots, u_{ij}, \ldots)$ are the weights, and x^i is the i^{th} coordinate of x. This MLP could be viewed in the modular form suggested by Fig. 1. Creating this MLP in **Torch** is just writing in C++ few lines describing this simple graph. Note that creating this MLP in **Torch** is like creating a new module: you can use it afterward to describe more complex machines. For example, if you need to create a mixture of experts [7], you just have to create a "mixer" which would combine several MLP-experts and an MLP-gater, as suggested by Fig. 2. Creating complex machines is thus a very simple process, especially if you consider the large number of modules already available in **Torch**. For instance, creating a radial basis function neural network,

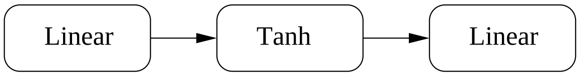

Figure 1: Modular view of the function $f_w(x) = v_0 + \sum_{j=1}^{N} v_j \tanh(u_{j0} + \sum_{i=1}^{d} u_{ji} x^i)$

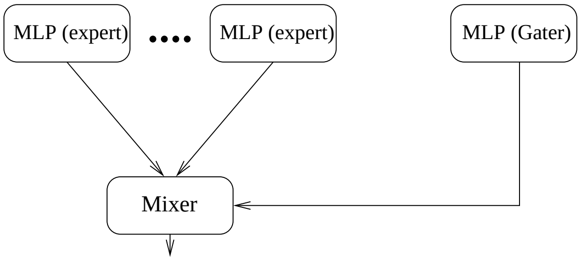

Figure 2: Modular view of a mixture of MLP

a time delay neural network (TDNN), or a convolutional neural network (such as *LeNet* [10]) takes only a few lines of C++ code!

Training a gradient machine is also quite easy: each cost function and each "trainer" is a module. Just select a cost function (such as mean-squared error or maximum likelihood) and give it to a trainer (for example a *stochastic* gradient trainer).

3.2 Support Vector Machines

Support Vector Machines (SVM), proposed in [14] have been applied to many classification problems, yielding good performance compared to other algorithms [5, 12]. For classification tasks, the decision function is of the form

$$y = \text{sign}\left(\sum_{i=1}^{T} y_i \alpha_i K(x, x_i) + b\right)$$

where $x \in \mathbb{R}^d$ is the d-dimensional input vector of a test example, $y \in \{-1, 1\}$ is a class label, x_i is the input vector for the i^{th} training example, y_i is its associated class label, T is the number of training examples, $K(x, x_i)$ is a *kernel* function and $\alpha = \{\alpha_1, \dots, \alpha_T\}$ and b are the parameters of the model. Training an SVM consists in finding α that minimizes the objective function

$$Q(\alpha) = -\sum_{i=1}^{T} \alpha_i + \frac{1}{2} \sum_{i=1}^{T} \sum_{j=1}^{T} \alpha_i \alpha_j y_i y_j K(x_i, x_j)$$

subject to the constraints:

$$\sum_{i=1}^{T} \alpha_i y_i = 0 \quad \textit{and} \quad 0 \le \alpha_i \le C.$$

Many kernels are available in **Torch**, but the most used one is the Gaussian kernel:

$$k(x_i, x_j) = \exp\left(-\gamma^2 \|x_i - x_j\|^2\right) \quad (\gamma \in \mathbb{R}).$$

Once again, SVMs are implemented in a modular way: SVM for regression and classification are two different modules, and each kernel is a module. Give them to a "quadratic constrained" trainer and you can train your specific SVM. Note that the training algorithm used in **Torch** has been proposed in [8, 4], and is one of the fastest available algorithms.

3.3 Distributions

A **Torch** distribution is an object, such as a Gaussian, that can, for instance, compute the probability or density of a data, or the likelihood of a data set. The parameters of such a distribution can be trained using various training algorithms, such as Expectation-Maximization (EM) or Viterbi algorithms. In fact a **Torch** distribution is a particular case of a gradient machine, and thus could also be trained with a gradient descent method to optimize any criterion, or could also be mixed with other gradient machines, to create very complex machines. Many distributions exist in **Torch**, and we describe here only the two most common ones.

3.3.1 Gaussian Mixture Models

Gaussian mixture models with diagonal covariance matrices are often used in machine learning to represent any static distribution. For this reason, they have of course been implemented in **Torch**. The density evaluated at an example x given such a distribution is

$$p(x) = \sum_{n=1}^{N} w_n \cdot \mathcal{N}(x; \mu_n, \sigma_n)$$

where $\mathcal{N}(x; \mu_n, \sigma_n)$ is a Gaussian with mean $\mu_n \in \mathbb{R}^d$ where d is the number of features and with σ_n the diagonal of the covariance matrix $\Sigma_n \in \mathbb{R}^{d^2}$:

$$\mathcal{N}(x; \mu, \Sigma) = \frac{1}{(2\pi)^{\frac{d}{2}}\sqrt{|\Sigma|}} \exp\left(-\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu)\right)$$

3.3.2 Hidden Markov Models

Hidden Markov models (HMMs) [13] are one of the most used techniques to represent sequences (such as biological sequences, speech data, or handwritten data). Basically, an HMM can model the density $p(X)$ of sequences X using a factored representation based on a set of states which are represented by emission distributions $p(x_t|q_t = i)$, and a table of transition probabilities $P(q_t = i|q_{t-1} = j)$, (where q_t is the state at time t and x_t is the t$^{\text{th}}$ frame of X). HMMs are often trained by EM or its Viterbi approximation. In **Torch**, both of them have been implemented, as well as gradient descent. This enables the user to create HMMs with complex distributions, such as any kind of neural network.

Moreover, several classes have also been implemented in **Torch** in order to be able to solve connected word speech recognition tasks. A small vocabulary decoder is already available in **Torch** and a large vocabulary decoder compatible with **Torch** will also be available soon on the web site.

3.4 Ensembles

Bagging [3] and boosting [6] are both "ensemble" algorithms: given a "weak" learning algorithm, they train several models using variations of the original dataset and then combine the obtained models by a weighted sum of their outputs. This kind of algorithms could be applied on almost any machine learning algorithm. Therefore, in **Torch**, you just have to decides the algorithm you want to "bag" or "boost" and then give it to a "bagging trainer" or a "boosting trainer".

4 Comparisons With Other Tools

Most of the machine learning algorithms implemented in **Torch** are often already available as standalone softwares. For instance HTK is a tool used to train hidden Markov models for speech recognition tasks; SVMLight and SVMTorch are tools often used to train support vector machines; and several packages are easily available to train neural networks. However, to the best of our knowledge, there are currently no unified platform that provide all these algorithms in the same efficient programming environment, letting the user easily compare various solutions (say an MLP and an SVM) on the same set of tasks, using the same measures of quality and techniques such as cross-validation or bootstrap to assess their relative performances. Moreover, thanks to its object oriented design, the same platform can be used to modify current algorithms or add new ones fairly easily.

In fact, several comparisons have been performed between **Torch** and these standalone softwares in order to compare (a) the precision and (b) the speed and memory requirements of the different platforms. For instance, a comparison between HTK and **Torch** on a speech recognition task using the benchmark database Numbers95 showed similar performances both in terms of training/decoding time and in terms of word error rates. Also, a comparison between **Torch** and SVMLight, which is a well-known package used to train support vector machines, showed similar results both in terms of performance and training time. Finally, several comparisons between **Torch** and various packages implementing multi-layer perceptrons and gradient descent were also performed, with in general similar but faster results obtained by **Torch**.

5 Conclusion

In this paper, we have presented **Torch**, a new machine learning software library freely available to the scientific community, which includes most popular algorithms and models such as multi-layer perceptrons, support vector machines, Gaussian mixture models, hidden Markov models, K nearest neighbors, Parzen windows, mixtures of experts, spatial and temporal convolutional neural networks (such as TDNNs and LeNet), Bagging, AdaBoost, Bayes classifiers*etc.* Being able to use all these algorithms in a simple yet unified framework enables researchers to scientifically compare them and easily enhance them. We strongly believe that providing such a platform to the community should help researchers to propose and develop novel solutions more quickly. Moreover, as the platform is open source, these novel algorithms can be quickly integrated in new versions of **Torch** and become available to the whole research community.

References

[1] C. Bishop. *Neural Networks for Pattern Recognition.* Clarendon Press, Oxford, 1995.

[2] Léon Bottou. *Une Approche théorique de l'Apprentissage Connexionniste: Applications à la Reconnaissance de la Parole.* PhD thesis, Université de Paris XI, Orsay, France, 1991.

[3] Leo Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1994.

[4] R. Collobert and S. Bengio. SVMTorch: Support vector machines for large-scale regression problems. *Journal of Machine Learning Research*, 1:143–160, 2001.

[5] C. Cortes and V. Vapnik. Support vector networks. *Machine Learning*, 20:273–297, 1995.

[6] Y. Freund and R. E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. In *Proceedings of the Second European Conference on Computational Learning Theory*, 1995.

[7] Robert A. Jacobs, Michael I. Jordan, Steven J. Nowlan, and Geoffrey E. Hinton. Adaptive mixtures of local experts. *Neural Computation*, 3(1):79–87, 1991.

[8] T. Joachims. Making large-scale support vector machine learning practical. In B. Schölkopf, C. Burges, and A. Smola, editors, *Advances in Kernel Methods.* The MIT Press, 1999.

[9] Y. LeCun. A theoretical framework for back-propagation. In D. Touretzky, G. Hinton, and T. Sejnowski, editors, *Proceedings of the 1988 Connectionist Models Summer School*, pages 21–28, CMU, Pittsburgh, Pa, 1988. Morgan Kaufmann.

[10] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998.

[11] Y. LeCun, L. Bottou, G. Orr, and K. Muller. Efficient backprop. In G. Orr and Muller K., editors, *Neural Networks: Tricks of the trade.* Springer, 1998.

[12] E. Osuna, R. Freund, and F. Girosi. Training support vector machines: an application to face detection. In *IEEE conference on Computer Vision and Pattern Recognition*, pages 130–136, San Juan, Puerto Rico, 1997.

[13] Laurence R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[14] V. N. Vapnik. *The nature of statistical learning theory.* Springer, second edition, 1995.